



Intel[®] Pentium[®] 4 and Xeon[™] Processor Optimization

Reference Manual

Copyright © 1999-2001 Intel Corporation

All Rights Reserved

Issued in U.S.A.

Order Number: 248966-03

World Wide Web: <http://developer.intel.com>

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

This Intel Pentium 4 Processor Optimization Reference Manual as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium 4 processor may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

* Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1999-2001.

Contents

Introduction

About This Manual	xxviii
Related Documentation	xxix
Notational Conventions	xxx

Chapter 1 Intel® Pentium® 4 Processor Overview

SIMD Technology and Streaming SIMD Extensions 2	1-2
Summary of SIMD Technologies	1-4
MMX Technology	1-4
Streaming SIMD Extensions	1-5
Streaming SIMD Extensions 2	1-5
Intel® NetBurst™ Micro-architecture	1-6
The Design Considerations of the Intel NetBurst Micro-architecture	1-7
Overview of the Intel NetBurst Micro-architecture Pipeline	1-8
The Front End	1-9
The Out-of-order Core	1-10
Retirement	1-11
Front End Pipeline Detail	1-12
Prefetching	1-12
Decoder	1-12
Execution Trace Cache	1-13
Branch Prediction	1-13
Branch Hints	1-15
Execution Core Detail	1-15

Instruction Latency and Throughput.....	1-16
Execution Units and Issue Ports	1-17
Caches.....	1-18
Data Prefetch	1-19
Loads and Stores	1-21
Store Forwarding.....	1-22
Chapter 2 General Optimization Guidelines	
Tuning to Achieve Optimum Performance.....	2-1
Tuning to Prevent Known Coding Pitfalls	2-2
General Practices and Coding Guidelines.....	2-3
Use Available Performance Tools	2-3
Optimize Performance Across Processor Generations	2-4
Optimize Branch Predictability	2-4
Optimize Memory Access	2-4
Optimize Floating-point Performance	2-5
Optimize Instruction Selection	2-5
Optimize Instruction Scheduling	2-6
Enable Vectorization	2-6
Coding Rules, Suggestions and Tuning Hints	2-6
Performance Tools.....	2-7
Intel® C++ Compiler	2-7
General Compiler Recommendations.....	2-8
VTune™ Performance Analyzer	2-9
Processor Generations Perspective	2-9
The CPUID Dispatch Strategy and Compatible Code Strategy	2-11
Branch Prediction	2-12
Eliminating Branches	2-12
Spin-Wait and Idle Loops.....	2-15
Static Prediction	2-15
Branch Hints	2-17
Inlining, Calls and Returns	2-18
Branch Type Selection	2-19

Loop Unrolling	2-20
Compiler Support for Branch Prediction	2-21
Memory Accesses	2-22
Alignment	2-22
Store Forwarding	2-25
Store-forwarding Restriction on Size and Alignment	2-26
Store-forwarding Restriction on Data Availability	2-30
Data Layout Optimizations	2-31
Stack Alignment	2-34
Aliasing Cases	2-35
Mixing Code and Data	2-36
Write Combining	2-37
Locality Enhancement	2-38
Prefetching	2-39
Hardware Instruction Fetching	2-39
Software and Hardware Cache Line Fetching	2-39
Cacheability instructions	2-40
Code	2-40
Improving the Performance of Floating-point Applications	2-41
Guidelines for Optimizing Floating-point Code	2-41
Floating-point Modes and Exceptions	2-43
Floating-point Exceptions	2-43
Floating-point Modes	2-45
Improving Parallelism and the Use of FXCH	2-49
x87 vs. SIMD Floating-point Trade-offs	2-50
Memory Operands	2-51
Floating-Point Stalls	2-51
x87 Floating-point Operations with Integer Operands	2-52
x87 Floating-point Comparison Instructions	2-52
Transcendental Functions	2-52
Instruction Selection	2-52
Complex Instructions	2-53

Use of the lea Instruction	2-53
Use of the inc and dec Instructions	2-54
Use of the shift and rotate Instructions	2-54
Integer and Floating-point Multiply	2-55
Integer Divide	2-55
Operand Sizes	2-55
Address Calculations	2-57
Clearing Registers	2-58
Compares	2-58
Floating Point/SIMD Operands	2-59
Prolog Sequences	2-60
Code Sequences that Operate on Memory Operands	2-60
Instruction Scheduling	2-61
Latencies and Resource Constraints	2-62
Spill Scheduling	2-62
Scheduling Rules for the Pentium 4 Processor Decoder	2-63
Vectorization	2-63
Miscellaneous	2-64
NOPs	2-64
Summary of Rules and Suggestions	2-65
User/Source Coding Rules	2-66
Assembly/Compiler Coding Rules	2-68
Tuning Suggestions	2-74

Chapter 3 Coding for SIMD Architectures

Checking for Processor Support of SIMD Technologies	3-2
Checking for MMX Technology Support	3-2
Checking for Streaming SIMD Extensions Support	3-3
Checking for Streaming SIMD Extensions 2 Support	3-4
Considerations for Code Conversion to SIMD Programming	3-6
Identifying Hot Spots	3-8
Determine If Code Benefits by Conversion to SIMD Execution	3-9
Coding Techniques	3-10

Coding Methodologies	3-10
Assembly.....	3-12
Intrinsics	3-13
Classes	3-14
Automatic Vectorization.....	3-15
Stack and Data Alignment	3-16
Alignment and Contiguity of Data Access Patterns	3-17
Using Padding to Align Data	3-17
Using Arrays to Make Data Contiguous	3-17
Stack Alignment For 128-bit SIMD Technologies.....	3-19
Data Alignment for MMX Technology	3-19
Data Alignment for 128-bit data	3-20
Compiler-Supported Alignment	3-21
Improving Memory Utilization	3-23
Data Structure Layout	3-23
Strip Mining	3-28
Loop Blocking	3-30
Instruction Selection	3-33
Tuning the Final Application	3-34
 Chapter 4 Optimizing for SIMD Integer Applications	
General Rules on SIMD Integer Code.....	4-2
Using SIMD Integer with x87 Floating-point	4-2
Using the EMMS Instruction	4-3
Guidelines for Using EMMS Instruction	4-4
Data Alignment	4-5
Data Movement Coding Techniques.....	4-5
Unsigned Unpack	4-5
Signed Unpack	4-6
Interleaved Pack with Saturation	4-7
Interleaved Pack without Saturation	4-9
Non-Interleaved Unpack	4-10
Extract Word	4-12

Insert Word	4-13
Move Byte Mask to Integer	4-15
Packed Shuffle Word for 64-bit Registers	4-17
Packed Shuffle Word for 128-bit Registers	4-18
Unpacking/interleaving 64-bit Data in 128-bit Registers	4-19
Data Movement	4-20
Conversion Instructions	4-20
Generating Constants	4-20
Building Blocks	4-21
Absolute Difference of Unsigned Numbers	4-22
Absolute Difference of Signed Numbers	4-22
Absolute Value	4-24
Clipping to an Arbitrary Range [high, low]	4-24
Highly Efficient Clipping	4-25
Clipping to an Arbitrary Unsigned Range [high, low]	4-27
Packed Max/Min of Signed Word and Unsigned Byte	4-28
Signed Word	4-28
Unsigned Byte	4-28
Packed Multiply High Unsigned	4-28
Packed Sum of Absolute Differences	4-28
Packed Average (Byte/Word)	4-29
Complex Multiply by a Constant	4-30
Packed 32*32 Multiply	4-31
Packed 64-bit Add/Subtract	4-31
128-bit Shifts	4-31
Memory Optimizations	4-31
Partial Memory Accesses	4-32
Increasing Bandwidth of Memory Fills and Video Fills	4-34
Increasing Memory Bandwidth Using the MOVDQ Instruction ...	4-35
Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page	4-35

	Increasing UC and WC Store Bandwidth by Using Aligned Stores	4-35
	Converting from 64-bit to 128-bit SIMD Integer	4-36
Chapter 5	Optimizing for SIMD Floating-point Applications	
	General Rules for SIMD Floating-point Code	5-1
	Planning Considerations	5-2
	Detecting SIMD Floating-point Support	5-2
	Using SIMD Floating-point with x87 Floating-point	5-3
	Scalar Floating-point Code	5-3
	Data Alignment	5-3
	Data Arrangement	5-4
	Vertical versus Horizontal Computation	5-4
	Data Swizzling	5-7
	Data Deswizzling	5-11
	Using MMX Technology Code for Copy or Shuffling Functions ..	5-15
	Horizontal ADD	5-15
	Use of cvtttps2pi/cvttss2si Instructions	5-19
	Flush-to-Zero Mode	5-19
Chapter 6	Optimizing Cache Usage for Intel Pentium 4 Processors	
	General Prefetch Coding Guidelines	6-2
	Prefetch and Cacheability Instructions	6-3
	Prefetch	6-4
	Software Data Prefetch	6-4
	Hardware Data Prefetch	6-5
	The Prefetch Instructions – Pentium 4 Processor Implementation	6-6
	Prefetch and Load Instructions	6-7
	Cacheability Control	6-8
	The Non-temporal Store Instructions	6-8
	Fencing	6-9
	Streaming Non-temporal Stores	6-9

Memory Type and Non-temporal Stores	6-9
Write-Combining	6-10
Streaming Store Usage Models	6-11
Coherent Requests	6-11
Non-coherent requests.....	6-11
Streaming Store Instruction Descriptions.....	6-12
The fence Instructions	6-13
The sfence Instruction.....	6-13
The lfence Instruction.....	6-14
The mfence Instruction.....	6-14
The clflush Instruction	6-14
Memory Optimization Using Prefetch	6-16
Software-controlled Prefetch	6-16
Hardware Prefetch	6-16
Example of Latency Hiding with S/W Prefetch Instruction	6-17
Prefetching Usage Checklist.....	6-19
Prefetch Scheduling Distance	6-19
Prefetch Concatenation	6-21
Minimize Number of Prefetches.....	6-23
Mix Prefetch with Computation Instructions.....	6-26
Prefetch and Cache Blocking Techniques	6-28
Single-pass versus Multi-pass Execution	6-33
Memory Optimization using Non-Temporal Stores	6-36
Non-temporal Stores and Software Write-Combining.....	6-36
Cache Management	6-37
Video Encoder.....	6-37
Video Decoder	6-38
Conclusions from Video Encoder and Decoder Implementation.	6-38
Using Prefetch and Streaming-store for a Simple Memory Copy	6-38
TLB Priming	6-39
Optimizing the 8-byte Memory Copy.....	6-40

Chapter A Application Performance Tools

Intel Compilers.....	A-2
Code Optimization Options	A-3
Targeting a Processor (-Gn).....	A-3
Automatic Process Dispatch Support (-Qx[extensions] and Qax[extensions])	A-3
Vectorizer Switch Options	A-4
Prefetching	A-4
Loop Unrolling	A-4
Multithreading with OpenMP	A-5
Inline Expansion of Library Functions (-Oi, -Oi-).....	A-5
Floating-point Arithmetic Precision (-Op, -Op-, -Qprec, -Qprec_div, -Qpc, -Qlong_double)	A-5
Rounding Control Option (-Qrcd)	A-5
Interprocedural and Profile-Guided Optimizations	A-6
Interprocedural Optimization (IPO)	A-6
Profile-Guided Optimization (PGO)	A-6
VTune™ Performance Analyzer	A-7
Using Sampling Analysis for Optimization	A-7
Time-based Sampling	A-7
Event-based Sampling	A-9
Sampling Performance Counter Events	A-9
Call Graph Profiling	A-12
Call Graph Window	A-12
Static Code Analysis	A-14
Static Assembly Analysis	A-15
Code Coach Optimizations	A-15
Assembly Coach Optimization Techniques	A-18
Intel® Performance Library Suite	A-19
Benefits Summary	A-19
Libraries Architecture	A-20
Optimizations with the Intel Performance Library Suite	A-21

	Enhanced Debugger (EDB)	A-21
	Intel® Architecture Performance Training Center.....	A-22
Chapter B	Intel Pentium 4 Processor Performance Metrics	
	Pentium 4 Processor-Specific Terminology	B-1
	Bogus, Non-bogus, Retire	B-1
	Bus Ratio	B-2
	Replay.....	B-2
	Assist	B-2
	Tagging	B-3
	Metrics Descriptions and Categories.....	B-3
	Performance Metrics and Tagging Mechanisms.....	B-13
	Tags for replay_event	B-13
	Tags for front_end_event	B-14
	Tags for execution_event.....	B-15
	Counting Clocks	B-16
Chapter C	IA-32 Instruction Latency and Throughput	
	Overview.....	C-1
	Definitions.....	C-3
	Latency and Throughput.....	C-4
	Latency and Throughput with Register Operands	C-5
	Table Footnotes.....	C-14
	Latency and Throughput with Memory Operands.....	C-15
Chapter D	Stack Alignment	
	Stack Frames	D-1
	Aligned esp-Based Stack Frames	D-4
	Aligned ebp-Based Stack Frames	D-6
	Stack Frame Optimizations.....	D-9
	Inlined Assembly and ebx.....	D-9

Chapter E Mathematics of Prefetch Scheduling Distance

Simplified Equation	E-1
Mathematical Model for PSD	E-2
No Preloading or Prefetch	E-5
Compute Bound (Case: $T_c \geq T_l + T_b$)	E-7
Compute Bound (Case: $T_l + T_b > T_c > T_b$)	E-8
Memory Throughput Bound (Case: $T_b \geq T_c$)	E-9
Example	E-10

Index**Examples**

2-1	Assembly Code with an Unpredictable Branch	2-13
2-2	Code Optimization to Eliminate Branches	2-13
2-3	Eliminating Branch with CMOV Instruction	2-14
2-4	Use of <code>pause</code> Instruction	2-15
2-5	Pentium 4 Processor Static Branch Prediction Algorithm	2-16
2-6	Static Taken Prediction Example	2-16
2-7	Static Not-Taken Prediction Example	2-17
2-8	Loop Unrolling	2-21
2-9	Code That Causes Cache Line Split	2-24
2-10	Several Situations of Small Loads After Large Store	2-27
2-11	A Non-forwarding Example of Large Load After Small Store	2-28
2-12	A Non-forwarding Situation in Compiler Generated code	2-28
2-13	Two Examples to Avoid the Non-forwarding Situation in Example 2-12	2-28
2-14	Large and Small Load Stalls	2-29
2-15	An Example of Loop-carried Dependence Chain	2-31
2-16	Rearranging a Data Structure	2-31
2-17	Decomposing an Array	2-32
2-18	Dynamic Stack Alignment	2-34
2-19	Algorithm to Avoid Changing the Rounding Mode	2-47

2-20	False Dependencies Caused by Referencing Partial Registers	2-56
2-21	Recombining LOAD/OP Code into REG, MEM Form.....	2-61
2-22	Spill Scheduling Example Code	2-62
3-1	Identification of MMX Technology with <code>cpuid</code>	3-2
3-2	Identification of SSE with <code>cpuid</code>	3-3
3-3	Identification of SSE by the OS	3-4
3-4	Identification of SSE2 with <code>cpuid</code>	3-5
3-5	Identification of SSE2 by the OS	3-5
3-6	Simple Four-Iteration Loop	3-11
3-7	Streaming SIMD Extensions Using Inlined Assembly Encoding.....	3-12
3-8	Simple Four-Iteration Loop Coded with Intrinsics.....	3-13
3-9	C++ Code Using the Vector Classes	3-15
3-10	Automatic Vectorization for a Simple Loop.....	3-16
3-11	C Algorithm for 64-bit Data Alignment	3-20
3-12	AoS data structure	3-24
3-13	SoA data structure	3-24
3-14	AoS and SoA Code Samples	3-25
3-15	Hybrid SoA data structure	3-27
3-16	Pseudo-code Before Strip Mining.....	3-29
3-17	Strip Mined Code	3-30
3-18	Loop Blocking	3-31
3-19	Emulation of Conditional Moves	3-33
4-1	Resetting the Register between <code>__m64</code> and FP Data Types.....	4-4
4-2	Unsigned Unpack Instructions.....	4-6
4-3	Signed Unpack Code.....	4-7
4-4	Interleaved Pack with Saturation	4-9
4-5	Interleaved Pack without Saturation	4-10
4-6	Unpacking Two Packed-word Sources in a Non-interleaved Way.....	4-12
4-7	<code>pextrw</code> Instruction Code	4-13
4-8	<code>pinsrw</code> Instruction Code	4-14

4-9	Repeated pinsrw Instruction Code	4-15
4-10	pmovmskb Instruction Code	4-16
4-11	pshuf Instruction Code.....	4-18
4-12	Broadcast using 2 instructions.....	4-18
4-13	Swap using 3 instructions.....	4-19
4-14	Reverse using 3 instructions	4-19
4-15	Generating Constants.....	4-20
4-16	Absolute Difference of Two Unsigned Numbers.....	4-22
4-17	Absolute Difference of Signed Numbers	4-23
4-18	Computing Absolute Value	4-24
4-19	Clipping to a Signed Range of Words [high, low]	4-25
4-20	Clipping to an Arbitrary Signed Range [high, low]	4-26
4-21	Simplified Clipping to an Arbitrary Signed Range.....	4-26
4-22	Clipping to an Arbitrary Unsigned Range [high, low]	4-27
4-23	Complex Multiply by a Constant.....	4-30
4-24	A Large Load after a Series of Small Stores (Penalty).....	4-32
4-25	Accessing Data without Delay	4-33
4-26	A Series of Small Loads after a Large Store	4-33
4-27	Eliminating Delay for a Series of Small Loads after a Large Store.....	4-34
5-1	Pseudocode for Horizontal (xyz, AoS) Computation	5-6
5-2	Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation....	5-7
5-3	Swizzling Data	5-8
5-4	Swizzling Data Using Intrinsics.....	5-9
5-5	Deswizzling Single-Precision SIMD Data	5-11
5-6	Deswizzling Data Using the movhlps and shuffle Instructions ..	5-13
5-7	Deswizzling Data 64-bit Integer SIMD Data	5-14
5-8	Using MMX Technology Code for Copying or Shuffling.....	5-15
5-9	Horizontal Add Using movhlps/movhlps.....	5-17
5-10	Horizontal Add Using Intrinsics with movhlps/movhlps	5-18
6-1	Pseudo-code for Using cflush.....	6-15
6-2	Prefetch Scheduling Distance	6-20

6-3	Using Prefetch Concatenation.....	6-22
6-4	Concatenation and Unrolling the Last Iteration of Inner Loop ...	6-22
6-5	Spread Prefetch Instructions	6-27
6-6	Data Access of a 3D Geometry Engine without Strip-mining	6-31
6-7	Data Access of a 3D Geometry Engine with Strip-mining	6-32
6-8	Basic Algorithm of a Simple Memory Copy	6-39
6-9	An Optimized 8-byte Memory Copy.....	6-40

Figures

1-1	Typical SIMD Operations	1-2
1-2	SIMD Instruction Register Usage	1-3
1-3	The Intel® NetBurst™ Micro-architecture	1-9
1-4	Execution Units and Ports in the Out-Of-Order Core	1-17
2-1	Cache Line Split in Accessing Elements in a Array	2-24
2-2	Size and Alignment Restrictions in Store Forwarding	2-26
3-1	Converting to Streaming SIMD Extensions Chart	3-7
3-2	Hand-Coded Assembly and High-Level Compiler Performance Trade-offs	3-11
3-3	Loop Blocking Access Pattern	3-32
4-1	PACKSSDW mm, mm/mm64 Instruction Example.....	4-8
4-2	Interleaved Pack with Saturation	4-8
4-3	Result of Non-Interleaved Unpack Low in MM0	4-11
4-4	Result of Non-Interleaved Unpack High in MM1	4-11
4-5	pextrw Instruction	4-13
4-6	pinsrw Instruction.....	4-14
4-7	pmovmskb Instruction Example.....	4-16
4-8	pshuf Instruction Example	4-17
4-9	PSADBW Instruction Example	4-29
5-1	Dot Product Operation.....	5-6
5-2	Horizontal Add Using movhlps/movlhps.....	5-16
6-1	Memory Access Latency and Execution Without Prefetch	6-18
6-2	Memory Access Latency and Execution With Prefetch	6-18
6-3	Prefetch and Loop Unrolling	6-23

6-4	Memory Access Latency and Execution With Prefetch	6-25
6-5	Cache Blocking – Temporally Adjacent and Non-adjacent Passes.....	6-29
6-6	Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops	6-30
6-7	Incorporating Prefetch into Strip-mining Code.....	6-33
6-8	Single-Pass Vs. Multi-Pass 3D Geometry Engines	6-35

Tables

1-1	Pentium 4 Processor Cache Parameters	1-18
2-1	Known Performance Issues in the Pentium 4 Processor	2-2
5-1	SoA Form of Representing Vertices Data.....	5-5
6-1	Prefetch Implementation: Pentium III and Pentium 4 Processors.....	6-7

Introduction

The Intel® Pentium® 4 Processor Optimization Reference Manual describes how to optimize software to take advantage of the performance characteristics of the newest Intel Pentium 4 processor. The optimizations described for the Pentium 4 processor will also apply to the future IA-32 processors based on the Intel® NetBurst™ micro-architecture.

The target audience for this manual includes software programmers and compiler writers. This manual assumes that the reader is familiar with the basics of the IA-32 architecture and has access to the three-volume set of manuals: *Intel® Architecture Software Developer's Manual: Volume 1, Basic Architecture*; *Volume 2, Instruction Set Reference*; and *Volume 3, System Programmer's Guide*.

When developing and optimizing software applications to achieve a high level of performance when running on IA-32 processors, a detailed understanding of IA-32 family of processors is often required; and in many cases, some level of knowledge on the micro-architecture of the newest IA-32 processors is also required.

This manual provides an overview of the Intel NetBurst micro-architecture, which is implemented in the Intel Pentium 4 processor and future IA-32 processors. This manual contains design guidelines for high-performance software applications, coding rules, and techniques for many aspects of code-tuning. These rules and techniques not only are useful to programmers, but are also applicable to compiler developers. This manual also includes instruction latency and throughput data for IA-32 instructions that pertain to the Pentium 4 processor.

The design guidelines that are discussed in this manual for developing high-performance software apply to current as well as to future IA-32 processors. Most of the coding rules and code optimization techniques based on the Intel NetBurst micro-architecture are also applicable to the P6 micro-architecture.

Tuning Your Application

Tuning an application for high performance on any IA-32 processor requires understanding and basic skills in the following areas:

- the IA-32 architecture
- C and Assembly language
- the hot-spot regions in your application that have significant impact on software performance
- the optimization capabilities of your compiler
- techniques to evaluate the application's performance.

The Intel VTune™ Performance Analyzer can help you analyze and locate any hot-spot regions in your applications. On the Pentium II, Pentium III, and Pentium 4 processors, this tool can monitor your application through a selection of performance monitoring events and analyze the performance event data that is gathered during code execution. This manual also describes information that can be gathered using the performance counters through Pentium 4 processor's performance monitoring events.

For VTune Performance Analyzer order information, see the web page:

<http://developer.intel.com>

About This Manual

Throughout this document, the reference “Pentium 4 processor” includes all processors based on the Intel NetBurst micro-architecture. Currently it refers to the Intel® Pentium 4 processor and Intel Xeon™ processor.

The manual consists of the following parts:

Introduction. Defines the purpose and outlines the contents of this manual.

Chapter 1: Pentium 4 Processor Overview. This chapter describes the new features of the Pentium 4 processor, including the architectural extensions to the IA-32 architecture and an overview of the Intel NetBurst micro-architecture.

Chapter 2: General Optimization Guidelines. Describes general code development and optimization techniques that apply to all applications designed to take advantage of the Intel NetBurst micro-architecture and high memory bandwidth.

Chapter 3: Coding for SIMD Architectures. Describes techniques and concepts for using the SIMD integer and SIMD floating-point instructions provided by the MMX^a technology, Streaming SIMD Extensions, and Streaming SIMD Extensions 2.

Chapter 4: Optimizing for SIMD Integer Applications. Provides optimization suggestions and common building blocks for applications that use the 64-bit and 128-bit SIMD integer instructions.

Chapter 5: Optimizing for SIMD Floating-point Applications. Provides optimization suggestions and common building blocks for applications that use the single-precision and double-precision SIMD floating-point instructions.

Chapter 6—Optimizing Cache Usage for Pentium 4 Processors. Describes how to use the `prefetch` instruction and cache control management instructions to optimize cache usage.

Appendix A—Application Performance Tools. Introduces several tools for analyzing and enhancing application performance without having to write assembly code.

Appendix B—Intel Pentium 4 Processor Performance Metrics. Provides a set of useful information that can be gathered using Pentium 4 processor's performance monitoring events. These performance metrics can help programmers determine how effectively an application is using the features of the Intel NetBurst micro-architecture.

Appendix C—IA-32 Instruction Latency and Throughput. Provides latency and throughput data for the IA-32 instructions. These data are specific to the implementation of the Pentium 4 processor.

Appendix D—Stack Alignment. Describes stack alignment conventions and techniques to optimize performance of accessing stack-based data.

Appendix E—The Mathematics of Prefetch Scheduling Distance. Discusses the optimum spacing to insert `prefetch` instructions and presents a mathematical model for determining the prefetch scheduling distance (PSD) for your application.

Related Documentation

For more information on the Intel architecture, specific techniques, and processor architecture terminology referenced in this manual, see the following documentation:

- *Intel® Architecture Optimization Reference Manual*, doc. number 245127
- *Pentium® Processor Family Developer's Manual*, Volumes 1, 2, and 3, doc. numbers 241428, 241429, and 241430
- *Intel® C++ Compiler User's Guide and Reference*
- *Intel® Fortran Compiler User's Guide and Reference*
- VTune™ Performance Analyzer online help
- *Intel® Architecture Software Developer's Manual*:
 - Volume 1: *Basic Architecture*, doc. number 243190
 - Volume 2: *Instruction Set Reference Manual*, doc. number 243191
 - Volume 3: *System Programmer's Guide*, doc. number 243192
- *Intel Processor Identification with the CPUID Instruction*, doc. number 241618.

Notational Conventions

This manual uses the following conventions:

<code>This type style</code>	Indicates an element of syntax, a reserved word, a keyword, a filename, instruction, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
<code>THIS TYPE STYLE</code>	Indicates a value, for example, TRUE, CONST1, or a variable, for example, A, B, or register names MMO through MM7. 1 indicates lowercase letter L in examples. 1 is the number 1 in examples. O is the uppercase O in examples. 0 is the number 0 in examples.
<i>This type style</i>	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
... (ellipses)	Indicate that a few lines of the code are omitted.
This type style	Indicates a hypertext link.

Intel® Pentium® 4 Processor Overview

1

This chapter gives an overview of the key features of the Intel® Pentium® 4 processor. This overview provides the background for understanding the coding recommendations described in detail in later chapters.

The key features of the Pentium 4 processor that enable high-performance applications are:

- Streaming SIMD Extensions 2 (SSE2) support
- Intel® NetBurst™ micro-architecture
- the implementation parameters of the Intel NetBurst micro-architecture in the Pentium 4 processor

The SSE2 is an architectural enhancement to the IA-32 architecture. The Intel NetBurst micro-architecture is a new micro-architecture implemented in the Pentium 4 processor. The implementation parameters of the Intel NetBurst micro-architecture in the Pentium 4 processor include:

- on-chip caches:
 - 8 KByte high-speed first-level data cache
 - 12K μ op Execution Trace Cache (TC)
 - 256 KByte unified 8-way second-level cache – Advanced Transfer Cache
- 400 MHz Intel NetBurst micro-architecture system bus, capable of delivering up to 3.2 GBytes per second of bandwidth.

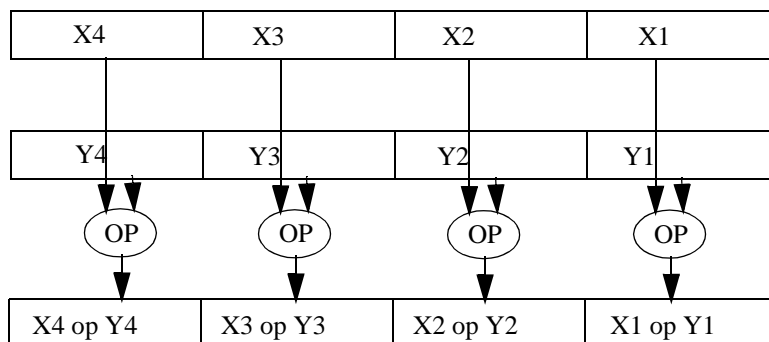
In addition to the above, this chapter introduces single-instruction, multiple-data (SIMD) technology. It also describes the theory of operation of the Pentium 4 processor with respect to the Intel NetBurst micro-architecture and the implementation characteristics of the Pentium 4 processor.

SIMD Technology and Streaming SIMD Extensions 2

One way to increase processor performance is to execute several computations in parallel, so that multiple computations are done with a single instruction. The way to achieve this type of parallel execution is to use the single-instruction, multiple-data (SIMD) computation technique.

[Figure 1-1](#) shows a typical SIMD computation. Here two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

Figure 1-1 Typical SIMD Operations



SIMD computations like those shown in [Figure 1-1](#) were introduced into the IA-32 architecture with the MMX™ technology. The MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers that are contained in a set of eight 64-bit registers called MMX registers (see [Figure 1-2](#)).

Figure 1-2 SIMD Instruction Register Usage

64-bit MMX Registers	128-bit XMM Registers
MM7	XMM7
MM6	XMM6
MM5	XMM5
MM4	XMM4
MM3	XMM3
MM2	XMM2
MM1	XMM1
MM0	XMM0

The Pentium III processor extended this initial SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). The Streaming SIMD Extensions allow SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be either in memory or in a set of eight 128-bit registers called the XMM registers (see Figure 1-2). The SSE also extended SIMD computational capability with additional 64-bit MMX instructions.

The Pentium 4 processor further extends the SIMD computation model with the introduction of the Streaming SIMD Extensions 2 (SSE2). The SSE2 also work with operands in either memory or in the XMM registers. The SSE2 extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in the SSE2 that can operate on two packed double-precision floating-point data elements, or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

The full set of IA-32 SIMD technologies (MMX technology, SSE, and SSE2) gives the programmer the ability to develop algorithms that can combine operations on packed 64- and 128-bit integer and single and double-precision floating-point operands.

This SIMD capability improves the performance of 3D graphics, speech recognition, image processing, scientific, and other multimedia applications that have the following characteristics:

- inherently parallel
- regular and recurring memory access patterns
- localized recurring operations performed on the data
- data-independent control flow.

The IA-32 SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. All SIMD instructions are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

The SSE2, SSE, and MMX technology are architectural extensions in the IA-32 Intel® architecture. All existing software continues to run correctly, without modification, on IA-32 microprocessors that incorporate these technologies. Existing software also runs correctly in the presence of new applications that incorporate these SIMD technologies.

The SSE and SSE2 instruction sets also introduced a set of cacheability and memory ordering instructions that can improve cache usage and application performance.

For more information on SSE2 instructions, including the cacheability and memory operation instructions, refer to the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1, Chapter 11 and Volume 2, Chapter 3 which are available at <http://developer.intel.com/design/pentium4/manuals/index.htm>.

Summary of SIMD Technologies

The paragraphs below summarize the new features of the three SIMD technologies (MMX technology, SSE, and SSE2) that have been added to the IA-32 architecture in chronological order.

MMX Technology

- Introduces 64-bit MMX registers.
- Introduces support for SIMD operations on packed byte, word, and doubleword integers.

The MMX instructions are useful for multimedia and communications software.

For more information on the MMX technology, refer to the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1, available at <http://developer.intel.com/design/pentium4/manuals/index.htm>.

Streaming SIMD Extensions

- Introduces 128-bit XMM registers.
- Introduces 128-bit data type with four packed single-precision floating-point operands.
- Introduces data prefetch instructions.
- Introduces non-temporal store instructions and other cacheability and memory ordering instructions.
- Adds extra 64-bit SIMD integer support.

The SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

For more information on the Streaming SIMD Extensions, refer to the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1, available at <http://developer.intel.com/design/pentium4/manuals/index.htm>.

Streaming SIMD Extensions 2

- Adds 128-bit data type with two packed double-precision floating-point operands.
- Adds 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Adds support for SIMD arithmetic on 64-bit integer operands.
- Adds instructions for converting between new and existing data types.
- Extends support for data shuffling.
- Extends support for cacheability and memory ordering operations.

The SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

For more information, refer to the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1, available at

<http://developer.intel.com/design/pentium4/manuals/index.htm>.

Intel® NetBurst™ Micro-architecture

The Pentium 4 processor is the first hardware implementation of a new micro-architecture, the Intel NetBurst micro-architecture. This section describes the key features of the Intel NetBurst micro-architecture and the details of its operation based on its implementation by the Pentium 4 processor. Additional Pentium 4 processor specific operational details, including instruction latencies, are given in [“IA-32 Instruction Latency and Throughput” in Appendix C](#). The information in this section provides the technical background to understand the optimization recommendations and coding rules that are discussed in Chapter 2 and the rest of this manual.

The Intel NetBurst micro-architecture is designed to achieve high performance for both integer and floating-point computations at very high clock rates. It supports the following features:

- hyper pipelined technology to enable high clock rates and frequency headroom to well above 1 GHz
- rapid execution engine to reduce the latency of basic integer instructions
- high-performance, quad-pumped bus interface to the 400 MHz Intel NetBurst micro-architecture system bus.
- rapid execution engine to reduce the latency of basic integer instructions
- out-of-order speculative execution to enable parallelism
- superscalar issue to enable parallelism
- hardware register renaming to avoid register name space limitations
- cache line sizes of 64 and 128 bytes
- hardware prefetch
- high-performance, quad-pumped bus interface to the Intel NetBurst micro-architecture system bus.

The Design Considerations of the Intel NetBurst Micro-architecture

The design goals of Intel NetBurst micro-architecture are: (a) to execute both the legacy IA-32 code and applications based on single-instruction, multiple-data (SIMD) technology at high processing rates; (b) to operate at high clock rates, and to scale to higher performance and clock rates in the future. To accomplish these design goals, the Intel NetBurst micro-architecture has many advanced features and improvements over the P6 micro-architecture.

To enable high performance and highly scalable clock rates, the major design considerations of the Intel NetBurst micro-architecture are as follows:

- It uses a deeply pipelined design to enable high clock rates with different parts of the chip running at different clock rates, some faster and some slower than the nominally-quoted clock frequency of the processor. The Intel NetBurst micro-architecture allows the Pentium 4 processor to achieve significantly higher clock rates as compared with the Pentium III processor. These clock rates will achieve well above 1 GHz.
- Its pipeline provides high performance by optimizing for the common case of frequently executed instructions. This means that the most frequently-executed instructions in common circumstances (such as a cache hit) are decoded efficiently and executed with short latencies, such that frequently encountered code sequences are processed with high throughput.
- It employs many techniques to hide stall penalties. Among these are parallel execution, buffering, and speculation. Furthermore, the Intel NetBurst micro-architecture executes instructions dynamically and out-of-order, so the time it takes to execute each individual instruction is not always deterministic. Performance of a particular code sequence may vary depending on the state the machine was in when that code sequence was entered.

Because of the complexity and subtlety of the Intel NetBurst micro-architecture, Chapter 2 of this document recommends what optimizations to use and what situations to avoid, and gives a sense of relative priority, but typically it does not absolutely quantify expected benefits and penalties. While this was more feasible with earlier in-order micro-architectures, this is no longer possible.

The following sections provide detailed description of the Intel NetBurst micro-architecture.

Overview of the Intel NetBurst Micro-architecture Pipeline

The pipeline of the Intel NetBurst micro-architecture contain three sections:

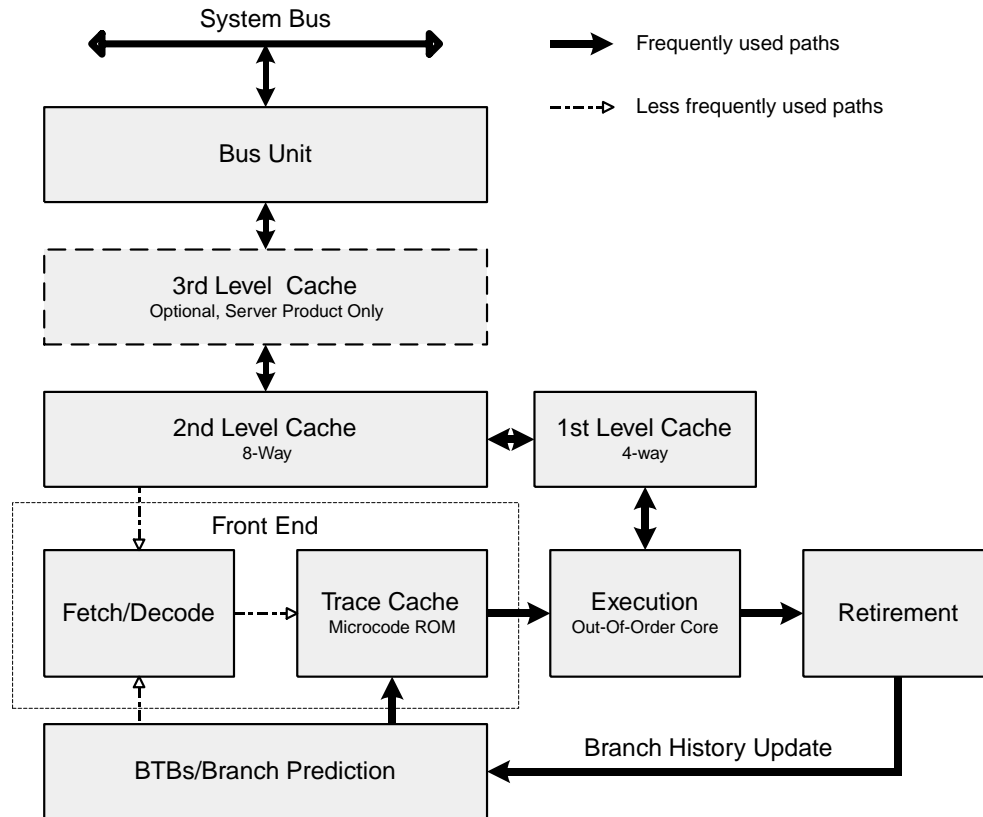
- the in-order issue front end
- the out-of-order superscalar execution core
- the in-order retirement unit.

The front end supplies instructions in program order to the out-of-order core. It fetches and decodes IA-32 instructions. The decoded IA-32 instructions are translated into μ ops. The front end's primary job is to feed a continuous stream of μ ops to the execution core in original program order.

The core can then issue multiple μ ops per cycle, and aggressively reorder μ ops so that those μ ops, whose inputs are ready and have execution resources available, can execute as soon as possible. The retirement section ensures that the results of execution of the μ ops are processed according to original program order and that the proper architectural states are updated.

[Figure 1-3](#) illustrates a block diagram view of the major functional units associated with the Intel NetBurst micro-architecture pipeline. The subsections that follow [Figure 1-3](#) provide an overview of each of the three sections in the pipeline.

Figure 1-3 The Intel® NetBurst™ Micro-architecture



The Front End

The front end of the Intel NetBurst micro-architecture consists of two parts:

- fetch/decode unit
- execution trace cache.

The front end performs several basic functions:

- prefetches IA-32 instructions that are likely to be executed
- fetches instructions that have not already been prefetched

- decodes instructions into micro-operations
- generates microcode for complex instructions and special-purpose code
- delivers decoded instructions from the execution trace cache
- predicts branches using highly advanced algorithm.

The front end of the Intel NetBurst micro-architecture is designed to address some of the common problems in high-speed, pipelined microprocessors. Two of these problems contribute to major sources of delays:

- the time to decode instructions fetched from the target
- wasted decode bandwidth due to branches or branch target in the middle of cache lines.

The execution trace cache addresses both of these problems by storing decoded IA-32 instructions. Instructions are fetched and decoded by a translation engine. The translation engine builds the decoded instruction into sequences of μ ops called traces, which are stored in the trace cache. The execution trace cache stores these micro-ops in the path of program execution flow, where the results of branches in the code are integrated into the same cache line. This increases the instruction flow from the cache and makes better use of the overall cache storage space since the cache no longer stores instructions that are branched over and never executed. The trace cache can deliver up to 3 μ ops per clock to the core.

The execution trace cache and the translation engine have cooperating branch prediction hardware. Branch targets are predicted based on their linear address using branch prediction logic and fetched as soon as possible. Branch targets are fetched from the execution trace cache if they are cached there, otherwise they are fetched from the memory hierarchy. The translation engine's branch prediction information is used to form traces along the most likely paths.

The Out-of-order Core

The core's ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one μ op is delayed while waiting for data or a contended resource, other μ ops that appear later in the program order may proceed around it. The processor employs several buffers to smooth the flow of μ ops. This implies that when one portion of the entire processor

pipeline experiences a delay, that delay may be covered by other operations executing in parallel (for example, in the core) or by the execution of μ ops which were previously queued up in a buffer (for example, in the front end).

The delays described in this chapter must be understood in this context. The core is designed to facilitate parallel execution. It can dispatch up to six μ ops per cycle through the issue ports pictured in [Figure 1-4, page 1-17](#). Note that six μ ops per cycle exceeds the trace cache and retirement μ op bandwidth. The higher bandwidth in the core allows for peak bursts of greater than 3 μ ops and to achieve higher issue rates by allowing greater flexibility in issuing μ ops to different execution ports.

Most execution units can start executing a new μ op every cycle, so that several instructions can be in flight at a time for each pipeline. A number of arithmetic logical unit (ALU) instructions can start two per cycle, and many floating-point instructions can start one every two cycles. Finally, μ ops can begin execution, out of order, as soon as their data inputs are ready and resources are available.

Retirement

The retirement section receives the results of the executed μ ops from the execution core and processes the results so that the proper architectural state is updated according to the original program order. For semantically-correct execution, the results of IA-32 instructions must be committed in original program order before it is retired. Exceptions may be raised as instructions are retired. Thus, exceptions cannot occur speculatively, they occur in the correct order, and the machine can be correctly restarted after an exception.

When a μ op completes and writes its result to the destination, it is retired. Up to three μ ops may be retired per cycle. The Reorder Buffer (ROB) is the unit in the processor which buffers completed μ ops, updates the architectural state in order, and manages the ordering of exceptions.

The retirement section also keeps track of branches and sends updated branch target information to the branch target buffer (BTB) to update branch history. [Figure 1-3](#) illustrates the paths that are most frequently executing inside the Intel NetBurst micro-architecture: an execution loop that interacts with multi-level cache hierarchy and the system bus.

The following sections describe in more detail the operation of the front end and the execution core. These detailed information of the Intel NetBurst micro-architecture provides the background for understanding the optimization techniques and using the instruction latency data that are documented in this manual.

Front End Pipeline Detail

The following information about the front end operation may be useful for tuning software with respect to prefetching, branch prediction, and execution trace cache operations.

Prefetching

The Intel NetBurst micro-architecture supports three prefetching mechanisms:

- the first is for instructions only
- the second is for data only
- the third is for code or data.

The first mechanism is hardware instruction fetcher that automatically prefetches instructions. The second is a software-controlled mechanism that fetches data into the caches using the prefetch instructions. The third is a hardware mechanism that automatically fetches data and instructions into the unified second-level cache.

The hardware instruction fetcher reads instructions along the path predicted by the BTB into the instruction streaming buffers. Data is read in 32-byte chunks starting at the target address. The second and third mechanisms will be described later.

Decoder

The front end of the Intel NetBurst micro-architecture has a single decoder that can decode instructions at the maximum rate of one instruction per clock. Some complex instructions must enlist the help of the microcode ROM. The decoder operation is connected to the execution trace cache discussed in the section that follows.

Execution Trace Cache

The execution trace cache (TC) is the primary instruction cache in the Intel NetBurst micro-architecture. The TC stores decoded IA-32 instructions, or μ ops. This removes decoding costs on frequently-executed code, such as template restrictions and the extra latency to decode instructions upon a branch misprediction.

In the Pentium 4 processor implementation, the TC can hold up to 12K μ ops and can deliver up to three μ ops per cycle. The TC does not hold all of the μ ops that need to be executed in the execution core. In some situations, the execution core may need to execute a microcode flow, instead of the μ op traces that are stored in the trace cache.

The Pentium 4 processor is optimized so that most frequently-executed IA-32 instructions come from the trace cache, efficiently and continuously, while only a few instructions involve the microcode ROM.

Branch Prediction

Branch prediction is very important to the performance of a deeply pipelined processor. Branch prediction enables the processor to begin executing instructions long before the branch outcome is certain. Branch delay is the penalty that is incurred in the absence of a correct prediction. For Pentium 4 processor, the branch delay for a correctly predicted instruction can be as few as zero clock cycles. The branch delay for a mispredicted branch can be many cycles; typically this is equivalent to the depth of the pipeline.

The branch prediction in the Intel NetBurst micro-architecture predicts all near branches, including conditional, unconditional calls and returns, and indirect branches. It does not predict far transfers, for example, far calls, irets, and software interrupts.

In addition, several mechanisms are implemented to aid in predicting branches more accurately and in reducing the cost of taken branches:

- dynamically predict the direction and target of branches based on the instructions' linear address using the branch target buffer (BTB)
- if no dynamic prediction is available or if it is invalid, statically predict the outcome based on the offset of the target: a backward branch is predicted to be taken, a forward branch is predicted to be not taken
- return addresses are predicted using the 16-entry return address stack

- traces of instructions are built across predicted taken branches to avoid branch penalties.

The Static Predictor. Once the branch instruction is decoded, the direction of the branch (forward or backward) is known. If there was no valid entry in the BTB for the branch, the static predictor makes a prediction based on the direction of the branch. The static prediction mechanism predicts backward conditional branches (those with negative displacement), such as loop-closing branches, as taken. Forward branches are predicted not taken.

To take advantage of the forward-not-taken and backward-taken static predictions, the code should be arranged so that the likely target of the branch immediately follows forward branches. See examples on branch prediction in “[Branch Prediction](#)” in Chapter 2.

Branch Target Buffer. Once branch history is available, the Pentium 4 processor can predict the branch outcome before the branch instruction is even decoded, based on a history of previously-encountered branches. It uses a branch history table and a branch target buffer (collectively called the BTB) to predict the direction and target of branches based on an instruction’s linear address. Once the branch is retired, the BTB is updated with the target address.

Return Stack. Returns are always taken, but since a procedure may be invoked from several call sites, a single predicted target will not suffice. The Pentium 4 processor has a Return Stack that can predict return addresses for a series of procedure calls. This increases the benefit of unrolling loops containing function calls. It also mitigates the need to put certain procedures inline since the return penalty portion of the procedure call overhead is reduced.

Even if the direction and target address of the branch are correctly predicted well in advance, a taken branch may reduce available parallelism in a typical processor, since the decode bandwidth is wasted for instructions which immediately follow the branch and precede the target, if the branch does not end the line and target does not begin the line. The branch predictor allows a branch and its target to coexist in a single trace cache line, maximizing instruction delivery from the front end.

Branch Hints

The Pentium 4 processor provides a feature that permits software to provide hints to the branch prediction and trace formation hardware to enhance their performance. These hints take the form of prefixes to conditional branch instructions. These prefixes have no effect for pre-Pentium 4 processor implementations. Branch hints are not guaranteed to have any effect, and their function may vary across implementations. However, since branch hints are architecturally visible, and the same code could be run on multiple implementations, they should be inserted only in cases which are likely to be helpful across all implementations.

Branch hints are interpreted by the translation engine, and are used to assist branch prediction and trace construction hardware. They are only used at trace build time, and have no effect within already-built traces. Directional hints override the static (forward-not-taken, backward-taken) prediction in the event that a BTB prediction is not available. Because branch hints increase code size slightly, the preferred approach to providing directional hints is by the arrangement of code so that

- forward branches that are more probable should be in the not-taken path, and
- backward branches that are more probable should be in the taken path. Since the branch prediction information that is available when the trace is built is used to predict which path or trace through the code will be taken, directional branch hints can help traces be built along the most likely path. See “[Branch Hints](#)” in Chapter 2 for branch hint coding recommendations.

Execution Core Detail

The execution core is designed to optimize overall performance by handling the most common cases most efficiently. The hardware is designed to execute the most frequent operations in the most common context as fast as possible, at the expense of less-frequent operations in rare context. Some parts of the core may speculate that a common condition holds to allow faster execution. If it does not, the machine may stall. An example of this pertains to store forwarding, see “[Store Forwarding](#)” later in this chapter. If a load is predicted to be dependent on a store, it gets its data from that store and tentatively proceeds. If the load turned out not to depend on the store, the load is delayed until the real data has been loaded from memory, then it proceeds.

Instruction Latency and Throughput

The superscalar, out-of-order core contains multiple execution hardware resources that can execute multiple μ ops in parallel. The core's ability to make use of available parallelism can be enhanced by:

- selecting IA-32 instructions that can be decoded into less than 4 μ ops and/or have short latencies
- ordering IA-32 instructions to preserve available parallelism by minimizing long dependence chains and covering long instruction latencies
- ordering instructions so that their operands are ready and their corresponding issue ports and execution units are free when they reach the scheduler.

This subsection describes port restrictions, result latencies, and issue latencies (also referred to as throughput) that form the basis for that ordering. Scheduling affects the way that instructions are presented to the core of the processor, but it is the execution core that reacts to an ever-changing machine state, reordering instructions for faster execution or delaying them because of dependence and resource constraints. Thus the ordering of instructions is more of a suggestion to the hardware.

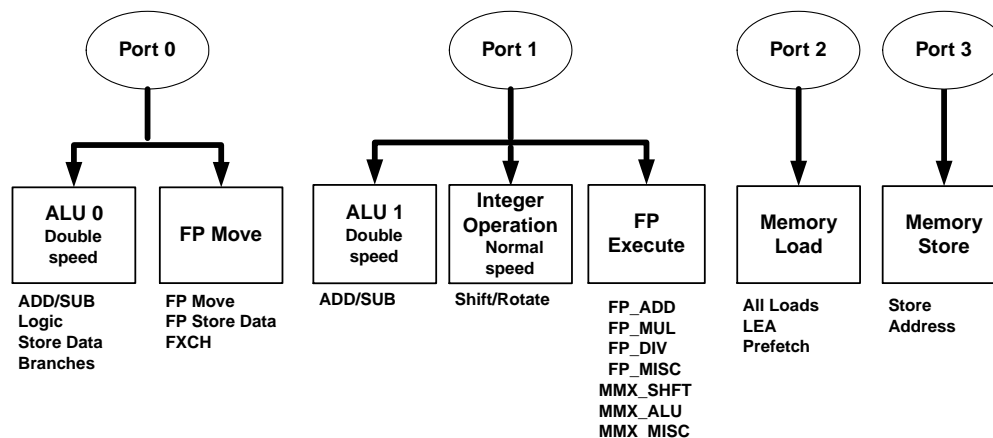
[“IA-32 Instruction Latency and Throughput” in Appendix C](#), lists the IA-32 instructions with their latency, their issue throughput, and in relevant cases, the associated execution units. Some execution units are not pipelined, such that μ ops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle.

The number of μ ops associated with each instruction provides a basis for selecting which instructions to generate. In particular, μ ops which are executed out of the microcode ROM, involve extra overhead. For the Pentium II and Pentium III processors, optimizing the performance of the decoder, which includes paying attention to the 4-1-1 sequence (instruction with four μ ops followed by two instructions each with one μ op) and taking into account the number of μ ops for each IA-32 instruction, was very important. On the Pentium 4 processor, the decoder template is not an issue. Therefore it is no longer necessary to use a detailed list of exact μ op count for IA-32 instructions. Commonly used IA-32 instruction which consists of four or less μ ops are shown in [“IA-32 Instruction Latency and Throughput” in Appendix C](#), with information on what execution units are associated with these instructions.

Execution Units and Issue Ports

Each cycle, the core may dispatch μ ops to one or more of the four issue ports. At the micro-architectural level, store operations are further divided into two parts: store data and store address operations. The four ports through which μ ops are dispatched to various execution units and to perform load and store operations are shown in [Figure 1-4](#). Some ports can dispatch two μ ops per clock because the execution unit for that μ op executes at twice the speed, and those execution units are marked “Double speed.”

Figure 1-4 Execution Units and Ports in the Out-Of-Order Core



Note:

FP_ADD refers to x87 FP, and SIMD FP add and subtract operations
 FP_MUL refers to x87 FP, and SIMD FP multiply operations
 FP_DIV refers to x87 FP, and SIMD FP divide and square-root operations
 MMX_ALU refers to SIMD integer arithmetic and logic operations
 MMX_SHFT handles Shift, Rotate, Shuffle, Pack and Unpack operations
 MMX_MISC handles SIMD reciprocal and some integer operations

Port 0. In the first half of the cycle, port 0 can dispatch either one floating-point move μ op (including floating-point stack move, floating-point exchange or floating-point store data), or one arithmetic logical unit (ALU) μ op (including arithmetic, logic or store data). In the second half of the cycle, it can dispatch one similar ALU μ op.

Port 1. In the first half of the cycle, port 1 can dispatch either one floating-point execution (all floating-point operations except moves, all SIMD operations) μ op or normal-speed integer (multiply, shift and rotate) μ op, or one ALU (arithmetic, logic or branch) μ op. In the second half of the cycle, it can dispatch one similar ALU μ op.

Port 2. Port 2 supports the dispatch of one load operation per cycle.

Port 3. Port 3 supports the dispatch of one store address operation per cycle.

Thus the total issue bandwidth can range from zero to six μ ops per cycle. Each pipeline contains several execution units. The μ ops are dispatched to the pipeline that corresponds to its type of operation. For example, an integer arithmetic logic unit and the floating-point execution units (adder, multiplier, and divider) share a pipeline.

Caches

The Intel NetBurst micro-architecture can support up to three levels of on-chip cache. Only two levels of on-chip caches are implemented in the Pentium 4 processor, which is a product for the desktop environment. The level nearest to the execution core of the processor, the first level, contains separate caches for instructions and data: a first-level data cache and the trace cache, which is an advanced first-level instruction cache. All other levels of caches are shared. The levels in the cache hierarchy are not inclusive, that is, the fact that a line is in level i does not imply that it is also in level $i+1$. All caches use a pseudo-LRU (least recently used) replacement algorithm. [Table 1-1](#) provides the parameters for all cache levels.

Table 1-1 Pentium 4 Processor Cache Parameters

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency, Integer/floating-point (clocks)	Write Update Policy
First	8KB	4	64	2/6	write through
TC	12K μ ops	8	N/A	N/A	N/A
Second	256KB	8	128 ¹	7/7	write back

¹ Two sectors per line, 64 bytes per sector

A second-level cache miss initiates a transaction across the system bus interface to the memory sub-system. The system bus interface supports using a scalable bus clock and achieves an effective speed that quadruples the speed of the scalable bus clock. It takes

on the order of 12 processor cycles to get to the bus and back within the processor, and 6-12 bus cycles to access memory if there is no bus congestion. Each bus cycle equals several processor cycles. The ratio of processor clock speed to the scalable bus clock speed is referred to as bus ratio. For example, one bus cycle for a 100 MHz bus is equal to 15 processor cycles on a 1.50 GHz processor. Since the speed of the bus is implementation- dependent, consult the specifications of a given system for further details.

Data Prefetch

The Pentium 4 processor has two mechanisms for prefetching data: a software-controlled prefetch and an automatic hardware prefetch.

Software-controlled prefetch is enabled using the four prefetch instructions introduced with Streaming SIMD Extensions (SSE) instructions. These instructions are hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is not intended for prefetching code. Using it can incur significant penalties on a multiprocessor system where code is shared.

Software-controlled data prefetch can provide optimal benefits in some situations, and may not be beneficial in other situations. The situations that can benefit from software-controlled data prefetch are the following:

- when the pattern of memory access operations in software allows the programmer to hide memory latency
- when a reasonable choice can be made of how many cache lines to fetch ahead of the current line being executed
- when an appropriate choice is made for the type of prefetch used. The four types of prefetches have different behaviors, both in terms of which cache levels are updated and the performance characteristics for a given processor implementation. For instance, a processor may implement the non-temporal prefetch by only returning data to the cache level closest to the processor core. This approach can have the following effects:
 - a) minimizing disturbance of temporal data in other cache levels
 - b) avoiding the need to access off-chip caches, which can increase the realized bandwidth compared to a normal load-miss, which returns data to all cache levels.

The situations that are less likely to benefit from software-controlled data prefetch are the following:

- In cases that are already bandwidth bound, prefetching tends to increase bandwidth demands, and thus not be effective.
- Prefetching too far ahead may cause eviction of cached data from the caches prior to actually being used in execution; not prefetching far enough ahead can reduce the ability to overlap memory and execution latencies.
- When the prefetch can only be usefully placed in locations where the likelihood of that prefetch's getting used is low. Prefetches consume resources in the processor and the use of too many prefetches can limit their effectiveness. Examples of this include prefetching data in a loop for a reference outside the loop, and prefetching in a basic block that is frequently executed, but which seldom precedes the reference for which the prefetch is targeted.

For more details on software prefetching see [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#).

Automatic hardware prefetch is a new feature in the Pentium 4 processor. It can bring cache lines into the unified second-level cache based on prior reference patterns. For more details on the automatic hardware prefetcher, see [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#).

Pros and Cons of Software and Hardware Prefetching. Software prefetching has the following characteristics:

- Handles irregular access patterns, which would not trigger the hardware prefetcher
- Handles prefetching of short arrays and avoids hardware prefetching's start-up delay before initiating the fetches
- Must be added to new code; does not benefit existing applications.

In comparison, hardware prefetching for Pentium 4 processor has the following characteristics:

- Works with existing applications
- Requires regular access patterns

- Has a start-up penalty before the hardware prefetcher triggers and begins initiating fetches. This has a larger effect for short arrays when hardware prefetching generates a request for data beyond the end of an array, which is not actually utilized. However, software prefetching can recognize and handle these cases by using fetch bandwidth to hide the latency for the initial data in the next array. The penalty diminishes if it is amortized over longer arrays.
- Avoids instruction and issue port bandwidth overhead.

Loads and Stores

The Pentium 4 processor employs the following techniques to speed up the execution of memory operations:

- speculative execution of loads
- reordering of loads with respect to loads and stores
- multiple outstanding misses
- buffering of writes
- forwarding of data from stores to dependent loads.

Performance may be enhanced by not exceeding the memory issue bandwidth and buffer resources provided by the machine. Up to one load and one store may be issued each cycle from the memory port's reservation stations. In order to be dispatched to the reservation stations, there must be a buffer entry available for that memory operation. There are 48 load buffers and 24 store buffers. These buffers hold the μ op and address information until the operation is completed, retired, and deallocated.

The Pentium 4 processor is designed to enable the execution of memory operations out of order with respect to other instructions and with respect to each other. Loads can be carried out speculatively, that is, before all preceding branches are resolved. However, speculative loads cannot cause page faults. Reordering loads with respect to each other can prevent a load miss from stalling later loads. Reordering loads with respect to other loads and stores to different addresses can enable more parallelism, allowing the machine to execute more operations as soon as their inputs are ready. Writes to memory are always carried out in program order to maintain program correctness.

A cache miss for a load does not prevent other loads from issuing and completing. The Pentium 4 processor supports up to four outstanding load misses that can be serviced either by on-chip caches or by memory.

Store buffers improve performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or cache is complete. Writes are generally not on the critical path for dependence chains, so it is often beneficial to delay writes for more efficient use of memory-access bus cycles.

Store Forwarding

Loads can be moved before stores that occurred earlier in the program if they are not predicted to load from the same linear address. If they do read from the same linear address, they have to wait for the store's data to become available. However, with store forwarding, they do not have to wait for the store to write to the memory hierarchy and retire. The data from the store can be forwarded directly to the load, as long as the following conditions are met:

- **Sequence:** The data to be forwarded to the load has been generated by a programmatically-earlier store, which has already executed.
- **Size:** the bytes loaded must be a subset of (including a proper subset, that is, the same) bytes stored.
- **Alignment:** the store cannot wrap around a cache line boundary, and the linear address of the load must be the same as that of the store.

General Optimization Guidelines

2

This chapter discusses general optimization techniques that can improve the performance of applications running on the Intel Pentium 4 processor. These techniques take advantage of the microarchitectural features of the Pentium 4 processor described in [Chapter 1, “Intel® Pentium® 4 Processor Overview”](#).

This chapter explains the optimization techniques both for those who use the Intel® C++ or Fortran Compiler and for those who use other compilers. The Intel compiler, which is specifically tuned for the Pentium 4 processor, provides the most of the optimization. For those not using the Intel C++ or Fortran Compiler, the assembly code tuning optimizations may be useful. The explanations are supported by coding examples.

Tuning to Achieve Optimum Performance

The most important factors in achieving optimum processor performance are:

- good branch prediction
- avoiding memory access stalls
- good floating-point performance
- instruction selection, including use of SIMD instructions
- instruction scheduling (to maximize trace cache bandwidth)
- vectorization.

The following sections describe important practices, tools, coding rules and recommendations associated with these factors that will aid in optimizing the performance on IA-32 processors.

Tuning to Prevent Known Coding Pitfalls

To produce program code that takes advantage of the strengths of the Intel NetBurst micro-architecture (as summarized in the previous section), performance tuning requires avoiding coding pitfalls that limit the performance of the target processor. This section lists several known coding pitfalls that could limit the performance of the Pentium 4 processor. Some of the coding pitfalls, such as the store-forwarding cases, also limit performance on Pentium III processors. This chapter provides recommendations and coding rules that help avoid them.

[Table 2-1](#) lists several known coding pitfalls that cause performance degradation in Pentium 4 processors. This table should be used as a check list for establishing a performance-tuning baseline for the Pentium 4 processor. For every issue, [Table 2-1](#) provides a reference to a section in this document, which describes in detail the causes of performance penalties and presents code examples, coding rules, and recommended solutions. Note that “aligned” here means that the address of the load is aligned with respect to the address of the store..

Table 2-1 Factors Affecting Performance in the Pentium 4 Processor

Factors Affecting Performance	Symptom	Example (if applicable)	Section Reference
Small, unaligned load after large store	Store-forwarding blocked	Example 2-10	Store Forwarding, Store-forwarding Restriction on Size and Alignment
Large load after small store; Load dword after store dword, store byte; Load dword, AND with 0xff after store byte	Store-forwarding blocked	Example 2-11, Example 2-12	Store Forwarding, Store-forwarding Restriction on Size and Alignment
Cache line splits	Access across cache line boundary	Example 2-9	Align data on natural operand size address boundaries
Integer shift and multiply latency	Longer latency than Pentium III processor		Use of the shift and rotate Instructions, Integer and Floating-point Multiply

continued

Table 2-1 Factors Affecting Performance in the Pentium 4 Processor (continued)

Factors Affecting Performance	Symptom	Example (if applicable)	Section Reference
Denormal inputs and outputs	Slows x87, SSE*, SSE2** floating-point operations		Floating-point Exceptions
Cycling more than 2 values of Floating-point Control Word	fldcw not optimized		Floating-point Modes

* Streaming SIMD Extensions (SSE)

** Streaming SIMD Extensions 2 (SSE2)

General Practices and Coding Guidelines

This section discusses the general guidelines that derive from the optimum performance factors listed in “[Tuning to Achieve Optimum Performance](#).” It also highlights key practices of using the performance tools.

The following is a summary of key practices of performance tools usage and general coding guidelines. Each heading is discussed in detail in subsequent sections in this chapter. The coding practices recommended under each heading and the bullets under each heading are listed in order of importance.

Use Available Performance Tools

- Current-generation compiler, such as the Intel C++ Compiler:
 - Set this compiler to produce code for the target processor implementation
 - Use the compiler switches for optimization and/or profile-guided optimization. These features are summarized in “[Intel® C++ Compiler](#)” and, in more detail, in the *Intel C++ Compiler User’s Guide and Reference*.
- Current-generation performance monitoring tools, such as VTune™ Performance Analyzer:
 - Identify performance issues, use event-based sampling, code coach and other analysis resource
 - Characterize performance gain.

Optimize Performance Across Processor Generations

- Use `cguid` dispatch strategy to deliver optimum performance for all processor generations.
- Use compatible code strategy to deliver optimum performance for Pentium 4 processor and future IA-32 processors.

Optimize Branch Predictability

- Improve branch predictability and optimize instruction prefetching by arranging code to be consistent with the static branch prediction assumptions: backward taken and forward not taken.
- Avoid mixing near and far calls and returns.
- Avoid implementing a call by pushing the return address and jumping to the target. The hardware can pair up call and return instructions to enhance predictability.
- Use the `pause` instruction in spin-wait loops.
- Inline functions according to coding recommendations.
- Eliminate branches.
- Avoid indirect calls.

Optimize Memory Access

- Observe store-forwarding constraints.
- Ensure proper data alignment to prevent data split across cache line. boundary. This includes stack and passing parameters.
- Avoid mixing code and data (self-modifying code).
- Choose data types carefully (see next bullet below) and avoid type casting.
- Employ data structure layout optimization to ensure efficient use of longer Pentium 4 processor cache lines.
- Use prefetching appropriately.
- Minimize use of global variables and pointers.
- Use the `const` modifier; use the `static` modifier for global variables.

- Use the following techniques to enhance locality: blocking, loop interchange, loop skewing.
- Use new cacheability instructions and memory-ordering behavior for Pentium 4 processor.

Optimize Floating-point Performance

- Avoid exceeding representable ranges during computation, since handling these cases can have a performance impact; however, do not use a larger precision format (double-extended floating point) unless required, since it increases memory size and bandwidth utilization.
- Use the optimized `fldcw` when possible, avoid changing floating-point control/status registers (rounding modes) between more than two values.
- Use efficient conversions, such as those that implicitly include a rounding mode, in order to avoid changing control/status registers.
- Take advantage of the SIMD capabilities of Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2) instructions; enable flush-to-zero mode and DAZ mode when using SSE and SSE2 instructions.
- Avoid denormalized input values, denormalized output values, and explicit constants that could cause denormal exceptions.
- Avoid excessive use of the `fxch` instruction.

Optimize Instruction Selection

- Avoid longer latency instructions: shifts, integer multiplies and divides. Replace them with alternate code sequences (e.g. adds instead of shifts, and shifts instead of multiplies).
- Use the `leaq` instruction and the full range of addressing modes to do address calculation.
- Some types of stores use more μ ops than others, try to use simpler store variants and/or reduce the number of stores.
- Avoid use of complex instructions that require more than 4 μ ops.

- Avoid instructions that unnecessarily introduce dependence-related stalls: `inc` and `dec` instructions, partial register operations (8/16-bit operands).
- Avoid use of `ah`, `bh`, and other higher 8-bits of the 16-bit registers, because accessing them requires a shift operation internally.
- Use `xor` and `pxor` instructions to clear registers and break dependencies.
- Use efficient approaches for performing comparisons.

Optimize Instruction Scheduling

- Consider latencies and resource constraints.
- Calculate store addresses as early as possible.
- Arrange load operations and store operations using the same address such that the load does not follow the store immediately, especially if the store depends on a long-latency operation.

Enable Vectorization

- Use the smallest possible data type, to enable more parallelism with the use of a longer vector.
- Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically-backward dependence.
- Avoid the use of conditionals.
- Keep induction (loop) variable expressions simple
- Avoid using pointers, try to replace pointers with arrays and indices.

Coding Rules, Suggestions and Tuning Hints

Chapter 2 includes rules, suggestions and hints. They are maintained in separately-numbered lists and are targeted for three audiences:

- those modifying the source to enhance performance (user/source rules)
- those writing assembly or compilers (assembly/compiler rules)

- those doing detailed performance tuning (tuning suggestions)

Coding recommendations are ranked by importance in two ways:

- Local impact (later on referred to as “impact”) is the difference that a recommendation makes to performance for a given instance with the priority marked as: H = high, M = medium, L = low.
- Generality – how frequently such instances occur across all application domains with the priority marked as: H = high, M = medium, L = low.

These rules are intentionally very approximate. They can vary depending on coding style, application domain, and other factors. The purpose of including high, medium and low priorities to each recommendation is to provide some hints to the degree of performance gain that one can expect if a recommendation is implemented. Because it is not possible to predict the frequency of occurrence of a code instance in applications, a priority hint cannot be directly correlated to application-level performance gain. However, in a few important cases where relevant application-level performance gain has been observed, a more quantitative characterization of application-level performance gain is provided for information only (See [“Store-forwarding Restriction on Size and Alignment”](#) and [“Instruction Selection”](#)). In places where there is no priority assigned, the impact or generality has been deemed inapplicable.

Performance Tools

Intel offers several tools that can facilitate your effort of optimizing your application’s performance.

Intel® C++ Compiler

Use the Intel C++ Compiler following the recommendations described here wherever possible. The Intel Compiler’s advanced optimization features provide good performance without the need to hand-tune assembly code. However, the following features may enhance performance even further:

- Inlined assembly
- Intrinsics, which have a one-to-one correspondence with assembly language instructions. (Refer to the “Intel C++ Intrinsics Reference” section of the *Intel C++ Compiler User’s Guide and Reference*.)
- C++ class libraries (Refer to the “Intel C++ Class Libraries for SIMD Operations Reference” section of the *Intel C++ Compiler User’s Guide and Reference*.)
- Vectorization, in conjunction with compiler directives (pragmas). (Refer to the “Compiler Vectorization Support and Guidelines” section of the *Intel C++ Compiler User’s Guide and Reference*.)

The Intel C++ Compiler can generate a single executable which uses features such as Streaming SIMD Extensions 2 to maximize performance on a Pentium 4 processor, but which will still execute correctly on older processors without such features. (Refer to the “Processor Dispatch Support” section in the *Intel C++ Compiler User’s Guide and Reference*.)

General Compiler Recommendations

Any compiler that has been extensively tuned for the Pentium 4 processor can be expected to match or outperform hand-coding in the general case. However, if particular performance problems are noted with the compiled code, some compilers (like the Intel C++ and Fortran Compilers) allow the coder to insert intrinsics or inline assembly, to exert greater control over what code is generated. If inlined assembly is used, the user should verify that the code generated to integrate the inline assembly is of good quality and yields good overall performance.

Default compiler switches are generally targeted for the common case. That is, an optimization may be made the default if it is beneficial for most programs. In the unlikely event that a performance problem is root-caused to a poor choice on the part of the compiler, using different switches for that compiler, or compiling that module with a different compiler may be fruitful alternatives.

Performance of compiler-generated code may vary from one compiler vendor to another. Intel’s C++ and Fortran Compilers are highly optimized for the Pentium 4 processor. You may find significant performance advantages to using this as your back-end compiler.

VTune™ Performance Analyzer

Where performance is of critical concern, use performance monitoring hardware and software tools to tune your application and its interaction with the hardware. The Pentium 4 processor provides counters which monitor a large number of performance-related events, effecting overall performance, branch prediction, the number and type of cache misses, and average trace length. The counters also provide information that helps resolve the coding pitfalls.

The VTune Performance Analyzer uses these counters to provide you with two kinds of feedback:

- an indication of a performance improvement from using a specific coding recommendation or microarchitectural feature
- information on whether a change in the program has improved or degraded performance with respect to a particular metric

Note that improving performance in one part of the machine does not necessarily bring significant gains to overall performance. In general, improving each component of performance should have an overall positive effect, although it is possible to degrade overall performance by improving performance for some particular metric.

Where appropriate, coding recommendations in this chapter include descriptions of the VTune analyzer events that provide measurable data of the performance gain achieved by following those recommendations. Refer to the VTune analyzer online help for instructions on how to use this tool.

The VTune analyzer events include a number of Pentium 4 processor performance metrics described in [Appendix B, “Intel Pentium 4 Processor Performance Metrics”](#).

Processor Generations Perspective

The Pentium 4 processor retains many of the features of the Pentium III processors, and adds a few new features. The majority of the coding recommendations for the Pentium 4 processor also apply to the Pentium III processors. However, there are notable differences, the most important of which are as follows:

- Instruction decode is now much less important. The scheduling concerns regarding the 4-1-1 template (instruction with four `µops` followed by two instructions with one `µop` each) no longer apply. The introduction of the trace cache (TC) means that the machine spends much less time decoding instructions.
- The loops should be exited with forward branches, if the extra branch incurs no added delay.
- Dependencies on partial register writes incurred large penalties on Pentium II and Pentium III processors. These penalties have been resolved by artificial dependencies between each partial register write. However, to avoid false dependences from partial register updates, full register updates and extended moves should be used.
- Some latencies have decreased; for example, these simple arithmetic operations are twice as fast: `add`, `sub`, `cmp`, `test`, `and`, `or`, `xor`, `neg`, `not`, `sahf`, `mov`.
- Some latencies have increased: shifts, rotates, integer multiplies, and moves from memory with sign extension are longer than before. Additional care must be taken regarding when to use the `lea` instruction. See the [“Use of the lea Instruction”](#) for specific recommendations.
- The `inc` and `dec` instructions should always be avoided. Using `add` and `sub` instructions instead of `inc` and `dec` instructions avoid data dependence and improve performance.
- Dependence-breaking support is added for the `pxor` instruction.
- Floating point register stack exchange instructions were free; now they are slightly more expensive due to issue restrictions.
- Writes and reads to the same location should now be spaced apart. This is especially true for writes that depend on long-latency instructions.
- Hardware prefetching may shorten the effective memory latency for data and instruction accesses.
- New cacheability instructions are available to streamline stores and manage cache utilization.
- Cache lines are 64 bytes on Pentium 4 processor (See [Table 1-1](#)), compared to cache line size of 32 bytes in Pentium II and Pentium III processors. Thus optimal prefetching should be done less often on Pentium 4 processors, and false sharing is more of an issue.

- The primary code size limit of interest is now imposed by the trace cache instead of the instruction cache.
- There may be a penalty when instructions with immediates requiring more than 16-bit signed representation are placed next to other instructions that use immediates.

Note that all of the memory-related optimization techniques for store-forwarding, data splits and alignments help Pentium 4 processor as well as Pentium III processors. Instruction selection using instruction latencies is one of the few instances where tuning for the Pentium 4 processor can slightly degrade performance of some code on the Pentium III processor.

The CPUID Dispatch Strategy and Compatible Code Strategy

Where optimum performance on all processor generations is desired, the application can take advantage of the `cputid` instruction to identify the processor generation and integrate processor-specific instructions (such as SSE2 instructions) into the source code where appropriate. The Intel C++ Compiler supports the integration of different versions of the code for each target processor within the same binary code. The selection of which code to execute at runtime is made based on the CPU identifier that is read with the `cputid` instruction. Binary code targeted for different processor generations can either be generated under the control of the programmer or automatically by the compiler.

For applications that must run on more than one generation of IA-32 processors, such as the Intel Pentium 4 and Pentium III processors, and where minimum binary code size and single code path is important, a compatible code strategy is the best. Using this strategy, only instructions common to the Pentium 4 and Pentium III processors are used in the source code. The programmer should optimize the application to achieve optimum performance on the Pentium 4 processor. This approach to optimization is also likely to deliver high performance on previous processor generations.

Branch Prediction

Branch optimizations have some of the greatest impact on performance. Understanding the flow of branches and improving the predictability of branches can increase the speed of your code significantly.

The basic kinds of optimizations that help branch prediction are:

- Keep code and data on separate pages (a very important item, see more details in the “[Memory Accesses](#)” section).
- Eliminate branches.
- Arrange code to be consistent with the static branch prediction algorithm.
- If it is not possible to arrange code, use branch direction hints where appropriate.
- Use the `pause` instruction in spin-wait loops.
- Inline functions and pair up calls and returns.
- Unroll as necessary so that repeatedly-executed loops have sixteen or fewer iterations, unless this causes an excessive code size increase.
- Separate branches so that they occur no more frequently than every three μ ops where possible.

Eliminating Branches

Eliminating branches improves performance due to:

- reducing the possibility of mispredictions
- reducing the number of required branch target buffer (BTB) entries; conditional branches, which are never taken, do not consume BTB resources.

There are four principal ways of eliminating branches:

- Arrange code to make basic blocks contiguous.
- Unroll loops, as discussed in the “[Loop Unrolling](#)” section.
- Use the `cmov` instruction.
- Use the `setcc` instruction.

Assembly/Compiler Coding Rule 1. (MH impact, H generality) *Arrange code to make basic blocks contiguous to eliminate unnecessary branches.*

Assembly/Compiler Coding Rule 2. (M impact, ML generality) *Use the `setcc` and `cmov` instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Also, do not use these instructions to eliminate all unpredictable conditional branches. Because using these instructions will incur execution overhead due to executing both paths of a conditional branch; Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch.*

Consider a line of C code that has a condition dependent upon one of the constants:

```
X = (A < B) ? CONST1 : CONST2;
```

This code conditionally compares two values, A and B. If the condition is true, x is set to CONST1; otherwise it is set to CONST2. An assembly code sequence equivalent to the above C code can contain branches that are not predictable if there are no correlation in the two values. [Example 2-1](#) shows the assembly code with unpredictable branches.

The unpredictable branches in [Example 2-1](#) can be removed with the use of the `setcc` instruction. [Example 2-2](#) shows an optimized code that does not have branches.

Example 2-1 Assembly Code with an Unpredictable Branch

```

cmp    A, B           ; condition
jge    L30            ; conditional branch
mov    ebx, CONST1    ; ebx holds X
jmp    L31            ; unconditional branch
L30:
mov    ebx, CONST2
L31:
```

Example 2-2 Code Optimization to Eliminate Branches

```

xor    ebx, ebx       ; clear ebx (X in the C code)
cmp    A, B
setge  bl             ; When ebx = 0 or 1
                        ; OR the complement condition
sub    ebx, 1         ; ebx=11...11 or 00...00
and    ebx, CONST3    ; CONST3 = CONST1-CONST2
add    ebx, CONST2    ; ebx=CONST1 or CONST2
```

The optimized code sets `ebx` to zero, then compares `A` and `B`. If `A` is greater than or equal to `B`, `ebx` is set to one. Then `ebx` is decreased and “and-ed” with the difference of the constant values. This sets `ebx` to either zero or the difference of the values. By adding `CONST2` back to `ebx`, the correct value is written to `ebx`. When `CONST2` is equal to zero, the last instruction can be deleted.

Another way to remove branches on Pentium II and following processors is to use the `cmov` and `fcmov` instructions. [Example 2-3](#) shows changing a test and branch instruction sequence using `cmov` and eliminating a branch. If the test sets the equal flag, the value in `ebx` will be moved to `eax`. This branch is data-dependent, and is representative of an unpredictable branch.

Example 2-3 Eliminating Branch with CMOV Instruction

```
test ecx, ecx
jne lh
mov  eax, ebx
lh:
; To optimize code, combine jne and mov into one cmovcc
; instruction that checks the equal flag
test    ecx, ecx    ; test the flags
cmoveq  eax, ebx    ; if the equal flag is set, move
                        ; ebx to eax - the lh: tag no longer needed
```

The `cmov` and `fcmov` instructions are available on the Pentium II and subsequent processors, but not on Pentium processors and earlier 32-bit Intel architecture processors. Be sure to check whether a processor supports these instructions with the `cpuid` instruction if an application needs to run on older processors as well. Code can often be arranged so that control can flow from one basic block to the next without having to execute a branch.

Spin-Wait and Idle Loops

The Pentium 4 processor introduces a new `pause` instruction which is architecturally a `nop` on all known IA-32 implementations. To the Pentium 4 processor, it acts as a hint that the code sequence is a spin-wait loop. Without a `pause` instruction in these loops, the Pentium 4 processor may suffer a severe penalty when exiting the loop because the processor detects a possible memory order violation. Inserting the `pause` instruction significantly reduces the likelihood of a memory order violation, improving performance. The Pentium 4 processor can execute a spin-wait loop using fewer resources and little power.

In [Example 2-4](#), the code is spinning until memory location `A` matches the value stored in the register `eax`. Such code sequences are common when protecting a critical section, in producer-consumer sequences, for barriers, or other synchronization.

Example 2-4 Use of `pause` Instruction

```
lock:  cmp eax, A
       jne loop
       ; code in critical section:
loop:  pause
       cmp eax, A
       jne loop
       jmp lock
```

Static Prediction

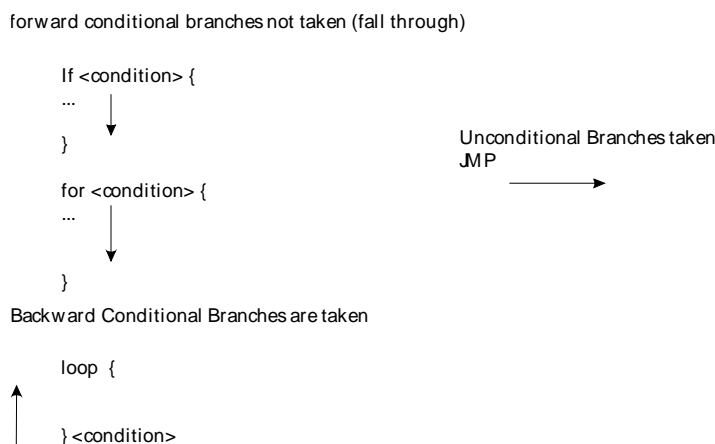
Branches that do not have a history in the BTB (see “[Branch Prediction](#)”) are predicted using a static prediction algorithm. The Pentium 4, Pentium III and Pentium II processors have the same static prediction algorithm. as follows:

- Predict unconditional branches to be taken.
- Predict backward conditional branches to be taken. This rule is suitable for loops.
- Predict forward conditional branches to be NOT taken.
- Predict indirect branches to be NOT taken.

Assembly/Compiler Coding Rule 3. (M impact, H generality) *Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target.*

[Example 2-5](#) illustrates the static branch prediction algorithm. The body of an `if-then` conditional is predicted to be executed.

Example 2-5 Pentium 4 Processor Static Branch Prediction Algorithm



Examples [2-6](#), [2-7](#) provide basic rules for the static prediction algorithm.

Example 2-6 Static Taken Prediction Example

```

Begin:  mov     eax, mem32
        and     eax, ebx
        imul    eax, edx
        shld    eax, 7
        jc     Begin

```

In [Example 2-6](#), the backward branch (`JC Begin`) is not in the BTB the first time through, therefore, the BTB does not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

Example 2-7 Static Not-Taken Prediction Example

```
mov     eax, mem32
and     eax, ebx
imul    eax, edx
shld    eax, 7
jc      Begin
mov     eax, 0
Begin:  call  Convert
```

The first branch instruction (`JC Begin`) in [Example 2-7](#) segment is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through.

The static prediction algorithm correctly predicts that the `Call Convert` instruction will be taken, even before the branch has any branch history in the BTB.

Branch Hints

The Pentium 4 processor provides a feature that permits the programmer to provide hints to the branch prediction and trace formation hardware to enhance their performance. These hints take the form of prefixes to any type of branch instruction. Branch hints are not guaranteed to have any effect, and their function may vary across implementations. On the Pentium 4 processor, branch hints are active only for relative conditional branches. However, since branch hints are architecturally visible to the decoder, they should be inserted only in cases which are likely to be helpful across all implementations or have significant benefit to the Pentium 4 processor implementation.

Branch hints are interpreted by the translation engine, and are used to assist branch prediction and trace construction hardware. They are only used at trace build time, and have no effect within built traces.

Directional hints override the static (forward-taken, backward-not-taken) prediction in the event that a BTB prediction is not available. Because branch hints increase code size slightly, the preferred approach to providing directional hints is by the arrangement of code so that forward branches are probably not taken and backward branches are. Since the branch prediction information, available when the trace is built, is used to predict which path or trace through the code will be taken, directional branch hints can help traces be built along the most likely path.

Use prefix 3E for taken and 2E for not taken conditional branches.

Assembly/Compiler Coding Rule 4. (L impact, MH generality) *Do not use directional branch hints if it is possible to position code to be consistent with the static branch prediction algorithm.*

In that case, there is no need to introduce a prefix, which increases code size.

Assembly/Compiler Coding Rule 5. *Use directional branch hints only in the case if the probability of the branch being taken in the prescribed direction is greater than 50%. Use code positioning to adhere to the static prediction algorithm wherever possible.*

There may be cases where predicting the initial direction differently from the typical direction may improve performance, but doing so is not recommended for long-term compatibility reasons.

Inlining, Calls and Returns

The return address stack mechanism augments the static and dynamic predictors to optimize specifically for calls and returns. It holds 16 entries, which is large enough to cover the call depth of most programs. If there is a chain of more than 16 nested calls, then more than 16 returns in rapid succession, performance may be degraded.

The trace cache maintains branch prediction information for calls and returns. As long as the trace with the call or return remains in the trace cache, and if the call and return targets remain unchanged, the depth limit of the return address stack described above will not impede performance.

To enable the use of the return stack mechanism, calls and returns must be matched up in pairs exactly. The likelihood of exceeding the stack depth in a manner that will impact performance is very low.

Assembly/Compiler Coding Rule 6. (MH impact, MH generality) *Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns.*

Calls and returns are expensive, therefore inlining can be used for these reasons:

- The parameter passing overhead can be eliminated.
- In a compiler, inlining a function can expose more opportunity for optimization.
- If the inlined routine contains branches, the additional context of the caller may improve branch prediction within the routine.
- A mispredicted branch can lead to larger performance penalties inside a small function than if that function is inlined.

Assembly/Compiler Coding Rule 7. (MH impact, MH generality) *Selectively inline a function where doing so decreases code size, or if the function is small and the call site is frequently executed.*

Assembly/Compiler Coding Rule 8. (H impact, M generality) *Do not inline a function if doing so increases the working set size beyond what will fit in the trace cache.*

Assembly/Compiler Coding Rule 9. (ML impact, ML generality) *If there are more than 16 nested calls and returns in rapid succession, then consider transforming the program, for example, with inline, to reduce the call depth.*

Assembly/Compiler Coding Rule 10. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred.*

Branch Type Selection

Counting loops can have a test and conditional branch at the top of the loop body or at the bottom.

Assembly/Compiler Coding Rule 11. (M impact, MH generality) *If the average number of total iterations is less than or equal to 100, use a forward branch to exit the loop.*

Indirect branches, such as switch statements, computed GOTOS or calls through pointers, can jump to an arbitrary number of locations. If the code sequence is such that the target destination of a branch goes to the same address most of the time, then the BTB will predict accurately most of the time. If, however, the target destination is not predictable, performance can degrade quickly.

User/Source Coding Rule 1. (L impact, L generality) *If some targets of an indirect branch are very predictable, correlate either with preceding branches or with the same branch, then convert the indirect branch into a tree where one or more indirect branches are preceded by conditional branches to those targets.*

Loop Unrolling

The benefits of unrolling loops are:

- Unrolling amortizes the branch overhead, since it eliminates branches and some of the code to manage induction variables.
- Unrolling allows you to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependence chain to expose the critical path.
- The Pentium 4 processor can correctly predict the exit branch for an inner loop that has 16 or fewer iterations, if that number of iterations is predictable and there are no conditional branches in the loop. Therefore, if the loop body size is not excessive, and the probable number of iterations is known, unroll inner loops until they have a maximum of 16 iterations. With Pentium III or Pentium II processors, do not unroll loops more than 4 iterations.

The potential costs of unrolling loops are:

- Excessive unrolling, or unrolling of very large loops can lead to increased code size. This can be harmful if the unrolled loop no longer fits in the trace cache (TC).
- Unrolling loops whose bodies contain branches increases the demands on the BTB capacity. If the number of iterations of the unrolled loop is 16 or less, the branch predictor should be able to correctly predict branches in the loop body that alternate direction.

Assembly/Compiler Coding Rule 12. (H impact, M generality) *Unroll small loops until the overhead of the branch and the induction variable accounts, generally, for less than about 10% of the execution time of the loop.*

Assembly/Compiler Coding Rule 13. (H impact, M generality) *Avoid unrolling loops excessively, as this may thrash the TC.*

Assembly/Compiler Coding Rule 14. (M impact, M generality) *Unroll loops that are frequently executed and that have a predictable number of iterations to reduce the number of iterations to 16 or fewer, unless this increases code size so that the working set no longer fits in the trace cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches).*

[Example 2-8](#) shows how unrolling enables other optimizations.

Example 2-8 Loop Unrolling

Before unrolling:

```
do i=1,100
  if (i mod 2 == 0) then a(i) = x
  else a(i) = y
enddo
```

After unrolling

```
do i=1,100,2
  a(i) = y
  a(i+1) = x
enddo
```

In this example, a loop that executes 100 times assigns *x* to every even-numbered element and *y* to every odd-numbered element. By unrolling the loop you can make both assignments each iteration, removing one branch in the loop body.

Compiler Support for Branch Prediction

Compilers can generate code that improves the efficiency of branch prediction in the Pentium 4 and Pentium III processors. The Intel C++ Compiler accomplishes this by:

- keeping code and data on separate pages
- using conditional move instructions to eliminate branches
- generate code that is consistent with the static branch prediction algorithm
- inlining where appropriate

- unrolling, if the number of iterations is predictable

Also, with profile-guided optimization, the Intel compiler can better lay out basic blocks to eliminate branches for the most frequently executed paths of a function, or at least improve their predictability. Thus the branch prediction need not be a concern at the source level. For more information, see the *Intel® C++ Compiler User's Guide and Reference*.

Memory Accesses

This section discusses guidelines for optimizing code and data memory accesses. The most important recommendations are:

- Align data, paying attention to data layout and stack alignment.
- Enable store forwarding.
- Place code and data on separate pages.
- Enhance data locality.
- Use prefetching and cacheability control instructions.
- Enhance code locality and align branch targets.
- Take advantage of write combining.

Alignment and forwarding problems are among the most common sources of large delays on the Pentium 4 processor.

Alignment

Alignment of data concerns all kinds of variables:

- dynamically allocated
- members of a data structure
- global or local variables
- parameters passed on the stack.

A misaligned data access can incur significant performance penalties. This is particularly true for cache line splits. The size of a cache line is 64 bytes in the Pentium 4 processor, and is 32 bytes in Pentium III and Pentium II processors. On the Pentium 4

processor, an access to data that are unaligned on 64-byte boundary lead to two memory accesses and requires several μ ops to be executed instead of one. Accesses that span either 16 byte or 64 byte boundaries are likely to incur a large performance penalty, since they are executed near retirement, and can incur stalls that are on the order of the depth of the pipeline.

Assembly/Compiler Coding Rule 15. (H impact, H generality) *Align data on natural operand size address boundaries*

For best performance, align data as follows:

- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned four byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.
- Align 80-bit data so that its base address is a multiple of sixteen.
- Align 128-bit data so that its base address is a multiple of sixteen.

A 64-byte or greater data structure or array should be aligned so that its base address is a multiple of 64. Sorting data in decreasing size order is one heuristic for assisting with natural alignment. As long as 16-byte boundaries (and cache lines) are never crossed, natural alignment is not strictly necessary, though it is an easy way to enforce this.

[Example 2-9](#) shows the type of code that can cause a cache line split. The code loads the addresses of two `dword` arrays. `029e70feh` is not a 4-byte-aligned address, so a 4-byte access at this address will get 2 bytes from the cache line this address is contained in, and 2 bytes from the cache line that starts at `029e7100h`. On processors with 64-byte cache lines, a similar cache line split will occur every 8 iterations.

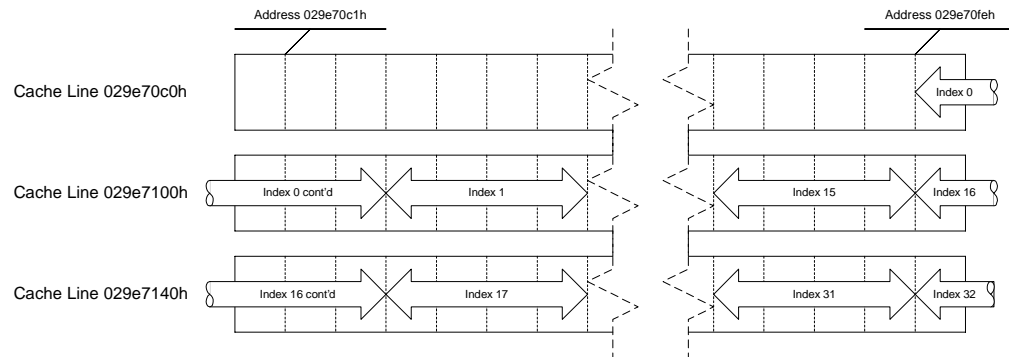
[Figure 2-1](#) illustrates the situation of accessing a data element that span across cache line boundaries.

Example 2-9 Code That Causes Cache Line Split

```

mov     esi, 029e70feh
mov     edi, 05be5260h
Blockmove:
mov     eax, DWORD PTR [esi]
mov     ebx, DWORD PTR [esi+4]
mov     DWORD PTR [edi], eax
mov     DWORD PTR [edi+4], ebx
add     esi, 8
add     edi, 8
sub     edx, 1
jnz     Blockmove

```

Figure 2-1 Cache Line Split in Accessing Elements in an Array

Alignment of code is much less of an issue for the Pentium 4 processor than for earlier processors. Alignment of branch targets to maximize bandwidth of fetching cached instructions is an issue only when not executing out of the trace cache.

Store Forwarding

The processor's memory system only sends stores to memory (including cache) after store retirement. However, store data can be forwarded from a store to a subsequent load from the same address to give a much shorter store-load latency.

There are two kinds of requirements for store forwarding. If these requirements are violated, store forwarding cannot occur, and the load must get its data from the cache (so the store must write its data back to the cache first). This incurs a penalty that is related to the pipeline depth. The first requirement pertains to the size and alignment of the store-forwarding data. This restriction is likely to have high impact to overall application performance. Typically, performance penalty due to violating this restriction can be prevented. Several examples of coding pitfalls that cause store-forwarding stalls and solutions to these pitfalls are discussed in detail in [“Store-forwarding Restriction on Size and Alignment”](#). The second requirement is the availability of data, discussed in [“Store-forwarding Restriction on Data Availability”](#).

A good practice is to eliminate redundant load operations, see some guidelines below:

Assembly/Compiler Coding Rule 16. (H impact, H generality) *Promote variables to registers where profitable.*

It may be possible to keep a temporary scalar variable in a register and never write it to memory. Generally, such a variable must not be accessible via indirect pointers. Moving a variable to a register eliminates all loads and stores of that variable, and thus eliminates potential problems associated with store forwarding. However, it also increases register pressure.

Assembly/Compiler Coding Rule 17. (MH impact, H generality) *Eliminate redundant loads.*

If a variable is known not to change between when it is stored and when it is used again, the register that was stored can be copied or used directly. If register pressure is too high, or an unseen function is called before the store and the second load, it may not be possible to eliminate the second load.

Assembly/Compiler Coding Rule 18. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible.*

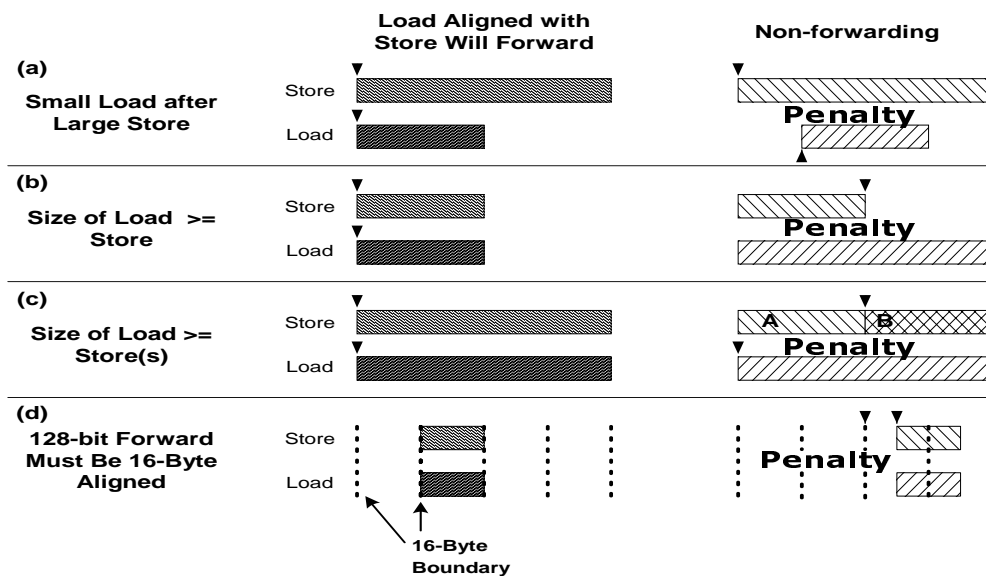
Parameter passing conventions may limit the choice of which parameters are passed in registers vs. on the stack. However, these limitations may be overcome if the compiler has control of the compilation of the whole binary, with whole-program optimization.

Store-forwarding Restriction on Size and Alignment

Data size and alignment restrictions for store-forwarding apply to Pentium 4 processor and previous generations of IA-32 processors. The performance penalty from violating store-forwarding restrictions was present in the Pentium II and Pentium III processors, but the penalty is larger on the Pentium 4 processor. It has been observed in several popular applications that the performance gain from not violating these restrictions is greater than 10%, at the application level on Pentium III processor as well as Pentium 4 processor. In general, the application-level performance gain will vary by application. This section describes this restriction in all its cases, and prescribes recommendations to prevent the non-forwarding penalty. Fixing this problem for the Pentium 4 processor also fixes the same kind of problem on Pentium II and Pentium III processors.

The size and alignment restrictions for store forwarding are illustrated in [Figure 2-2](#).

Figure 2-2 Size and Alignment Restrictions in Store Forwarding



Coding rules to help programmers satisfy size and alignment restrictions for store forwarding follow.

Assembly/Compiler Coding Rule 19. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data.*

Assembly/Compiler Coding Rule 20. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data.*

A load that forwards from a store must wait for the store's data to be written to the store buffer before proceeding, but other, unrelated loads need not wait.

Assembly/Compiler Coding Rule 21. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. The penalty for not doing this is much higher than the cost of the shifts.*

This is better than incurring the penalties of a failed store-forward.

Assembly/Compiler Coding Rule 22. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed.*

[Example 2-10](#) contains several store-forwarding situations when small loads follow large stores. The first three load operations illustrate the situations described in Rule 22. However, the last load operation gets data from store-forwarding without problem.

Example 2-10 Several Situations of Small Loads After Large Store

```
mov [EBP], 'abcd'
mov AL, [EBP]      ; not blocked - same alignment
mov BL, [EBP + 1]  ; blocked
mov CL, [EBP + 2]  ; blocked
mov DL, [EBP + 3]  ; blocked
mov AL, [EBP]      ; not blocked - same alignment
                  ; n.b. passes older blocked loads
```

[Example 2-11](#) illustrates a store-forwarding situation when a large load follows after several small stores. The data needed by the load operation cannot be forwarded because all of the data that needs to be forwarded is not contained in the store buffer. Avoid large loads after small stores to the same area of memory.

Example 2-11 A Non-forwarding Example of Large Load After Small Store

```

mov [EBP],      'a'
mov [EBP + 1],  'b'
mov [EBP + 2],  'c'
mov [EBP + 3],  'd'
mov EAX, [EBP]           ; blocked
; The first 4 small store can be consolidated into
; a single DWORD store to prevent this non-forwarding situation

```

[Example 2-12](#) illustrates a stalled store-forwarding situation that may appear in compiler generated code. Sometimes a compiler generates code similar to that shown in [Example 2-12](#) to handle spilled byte to the stack and convert the byte to an integer value.

Example 2-12 A Non-forwarding Situation in Compiler Generated code

```

mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
mov eax, DWORD PTR [esp+10h] ; Stall
and eax, 0xff                ; converting back to byte value

```

[Example 2-13](#) offers two alternatives to avoid the non-forwarding situation shown in [Example 2-12](#).

Example 2-13 Two Examples to Avoid the Non-forwarding Situation in Example 2-12

```

;A. Use movz instruction to avoid large load after small store, when
; spills are ignored
movz eax, bl                ; Replaces the last three instructions
                             ; in Example 2-12

;B. Use movz instruction and handle spills to the stack
mov DWORD PTR [esp+10h], 00000000h
mov BYTE PTR [esp+10h], bl
movz eax, BYTE PTR [esp+10h] ; not blocked

```

When moving data that is smaller than 64 bits between memory locations, 64- or 128-bit SIMD register moves are more efficient (if aligned) and can be used to avoid unaligned loads. Although floating-point registers allow the movement of 64 bits at a time, floating point instructions should not be used for this purpose, as data may be inadvertently modified.

As an additional example, consider the following cases in [Example 2-14](#). In the first case (A), there is a large load after a series of small stores to the same area of memory (beginning at memory address `mem`), and the large load will stall.

The `fld` must wait for the stores to write to memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

Example 2-14 Large and Small Load Stalls

```

;A. Large load stall
mov     mem, eax           ; store dword to address "mem"
mov     mem + 4, ebx       ; store dword to address "mem + 4"
fld     mem                ; load qword at address "mem", stalls

;B. Small Load stall
fstp    mem               ; store qword to address "mem"
mov     bx,mem+2           ; load word at address "mem + 2", stalls
mov     cx,mem+4           ; load word at address "mem + 4", stalls

```

In the second case ([Example 2-14](#), B), there is a series of small loads after a large store to the same area of memory (beginning at memory address `mem`), and the small loads will stall.

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible.

Store-forwarding Restriction on Data Availability

The value to be stored must be available before the load operation can be completed. If this restriction is violated, the execution of the load will be delayed until the data is available. This delay causes some execution resources to be used unnecessarily, and that can lead to some sizable but non-deterministic delays. However, the overall impact of this problem is much smaller than that from size and alignment requirement violations.

Assembly/Compiler Coding Rule 23. (H impact, M generality) *Space out loads from the store that forwards data to them. Note that placing intervening instructions between the load and store does not guarantee separation in time.*

The Pentium 4 processor predicts when loads are both, dependent on and get their data forwarded from, preceding stores. These predictions can significantly improve performance. However, if a load is scheduled too soon after the store it depends on, or more likely, if the generation of the data to be stored is delayed, there can be a significant penalty.

There are several cases where data is passed through memory, where the store may need to be separated from the load:

- spills, save and restore registers in a stack frame
- parameter passing
- global and volatile variables
- type conversion between integer and floating point
- some compilers do not analyze code that is inlined, forcing variables that are involved in the interface with inlined code to be in memory, creating more memory variables and preventing the elimination of redundant loads.

Assembly/Compiler Coding Rule 24. (ML impact, M generality) *If a routine is small, space apart the loads and stores that manage registers temporarily stored on the stack by re-loading the registers in the same order that they were stored; that is, replace pushes and pops with loads and stores, with the stores in the reverse order of pops.*

Assembly/Compiler Coding Rule 25. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, so as to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency*

instruction, e.g. *mul*, *div*. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain.

An example of a loop-carried dependence chain is shown in Example 2-15.

Example 2-15 An Example of Loop-carried Dependence Chain

```
for (i=0; i<MAX; i++) {  
    a[i] = b[i] * foo;  
    foo = a[i]/3;  
} // foo is a loop-carried dependence
```

Data Layout Optimizations

User/Source Coding Rule 2. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary.*

Align data by providing padding inside structures and arrays. Programmers can reorganize structures and arrays to minimize the amount of memory wasted by padding. However, compilers might not have this freedom. The C programming language, for example, specifies the order in which structure elements are allocated in memory. Section “[Stack and Data Alignment](#)” of Chapter 3, and [Appendix D, “Stack Alignment”](#), further defines the exact storage layout. [Example 2-16](#) shows how a data structure could be rearranged to reduce its size.

Example 2-16 Rearranging a Data Structure

```
struct unpacked { /* fits in 20 bytes due to padding */  
    int    a;  
    char   b;  
    int    c;  
    char   d;  
    int    e;  
}
```

continued

Example 2-16 Rearranging a Data Structure (continued)

```
struct packed { /* fits in 16 bytes */
    int    a, c, e;
    char   b, d;
}
```

Additionally, the longer cache line size for Pentium 4 processor can impact streaming applications (for example, multimedia) which reference and use data only once before discarding it. Data accesses which sparsely utilize the data within a cache line can result in less efficient utilization of system memory bandwidth. For example, arrays of structures can be decomposed into several arrays to achieve better packing, as shown in [Example 2-17](#).

Example 2-17 Decomposing an Array

```
struct { /* 1600 bytes */
    int  a, c, e;
    char b, d;
} array_of_struct [100];

struct { /* 1400 bytes */
    int  a[100], c[100], e[100];
    char b[100], d[100];
} struct_of_array;

struct { /* 1200 bytes */
    int a, c, e;
} hybrid_struct_of_array_ace[100];

struct { /* 200 bytes */
    char b, d;
} hybrid_struct_of_array_bd[100];
```

The efficiency of such optimizations depends on usage patterns. If the elements of the structure are all accessed together, but the access pattern of the array is random, then `array_of_struct` avoids unnecessary prefetch even though it wastes memory.

However, if the access pattern of the array exhibits locality, such as if the array index is being swept through, then the Pentium 4 processor prefetches data from `struct_of_array`, even if the elements of the structure are accessed together.

Moreover, when the elements of the structure are not accessed with equal frequency, such as when element `a` is accessed ten times more often than the other entries, then `struct_of_array` not only saves memory, but it also prevents fetching unnecessary data items `b`, `c`, `d`, and `e`.

Using `struct_of_array` also enables the use of the SIMD data types by the programmer and the compiler.

Note that `struct_of_array` can have the disadvantage of requiring more independent memory stream references. This can require the use of more prefetches, additional address generation calculations, as well as having a greater impact on DRAM page access efficiency. An alternative, `hybrid_struct_of_array` blends the two approaches. In this case, only 2 separate address streams are generated and referenced: 1 for `hybrid_struct_of_array_ac` and 1 for `hybrid_struct_of_array_bd`. This also prevents fetching unnecessary data, assuming the variables `a`, `c` and `e` are always used together; whereas the variables `b` and `d` would be also used together, but not at the same time as `a`, `c` and `e`. This hybrid approach ensures:

- simpler/fewer address generation than `struct_of_array`
- fewer streams, which reduces DRAM page misses
- use of fewer prefetches due to fewer streams
- efficient cache line packing of data elements that are used concurrently.

Assembly/Compiler Coding Rule 26. (H impact, M generality) *Try to arrange data structures such that they permit sequential access.*

If the data is arranged into set of streams, the automatic hardware prefetcher can prefetch data that will be needed by the application, reducing the effective memory latency. If the data is accessed in a non-sequential manner, the automatic hardware prefetcher cannot prefetch the data. The prefetcher can recognize up to eight concurrent streams. See [Chapter 6](#) for more information and the hardware prefetcher.

Memory coherence is maintained on 64-byte cache lines on Pentium 4, rather than earlier processors' 32-byte cache lines. This can increase the opportunity for false sharing.

User/Source Coding Rule 3. (M impact, L generality) *Beware of false sharing within 64-byte cache lines.*

Stack Alignment

The easiest way to avoid stack alignment problems is to keep the stack aligned at all times. For example, if a language only supports 8-bit, 16-bit, 32-bit, and 64-bit data quantities, but never uses 80-bit data quantities, the language can require the stack to always be aligned on a 64-bit boundary.

Assembly/Compiler Coding Rule 27. (H impact, M generality) *If 64-bit data is ever passed as a parameter or allocated on the stack, make sure that the stack is aligned to an 8-byte boundary.*

A routine that makes frequent use of 64-bit data can avoid stack misalignment by placing the code described in [Example 2-18](#) in the function prologue and epilogue.

Example 2-18 Dynamic Stack Alignment

```
prologue:
    subl    esp, 4           ; save frame ptr
    movl    [esp], ebp
    movl    ebp, esp        ; new frame pointer
    andl    ebp, 0xFFFFFFF0 ; aligned to 64 bits
    movl    [ebp], esp      ; save old stack ptr
    subl    esp, FRAME_SIZE ; allocate space
    ; ... callee saves, etc.
epilogue:
    ; ... callee restores, etc.
    movl    esp, [ebp]      ; restore stack ptr
    movl    ebp, [esp]      ; restore frame ptr
    addl    esp, 4
    ret
```

If for some reason it is not possible to align the stack for 64-bit, the routine should access the parameter and save it into a register or known aligned storage, thus incurring the penalty only once.

Aliasing Cases

There are several cases where addresses with a given stride will compete for some resource in the memory hierarchy. Note that first-level cache lines are 64 bytes and second-level cache lines are 128 bytes. Thus the least significant 6 or 7 bits are not considered in alias comparisons. The aliasing cases are listed below.

- 2K for data – map to the same first-level cache set (32 sets, 64-byte lines). There are 4 ways in the first-level cache, so if there are more than 4 lines that alias to the same 2K modulus in the working set, there will be an excess of first-level cache misses.
- 16K for data – will look the same to the store-forwarding logic. If there has been a store to an address which aliases with the load, the load will stall until the store data is available.
- 16K for code – can only be one of these in the trace cache at a time. If two traces whose starting addresses are 16K apart are in the same working set, the symptom will be a high trace cache miss rate. Solve this by offsetting one of the addresses by 1 or more bytes.
- 32K for code or data – map to the same second-level cache set (256 sets, 128-byte lines). There are 8 ways in the second-level cache, so if there are more than 8 lines that alias to the same 32K modulus in the working set, there will be an excess of second-level cache misses.
- 64K for data – can only be one of these in the first-level cache at a time. If a reference (load or store) occurs that has bits 0-15 of the linear address, which are identical to a reference (load or store) which is under way, then the second reference cannot begin until the first one is kicked out of the cache. Avoiding this kind of aliasing can lead to a factor of three speedup.

If a large number of data structures are in the same working set, accesses to aliased locations in those different data sets may cause cache thrashing and store forwarding problems. For example, if the code dynamically allocates many data 3KB structures, some memory allocators will return starting addresses for these structures which are on

4KB boundaries. For the sake of simplifying this discussion, suppose these allocations were made to consecutive 4KB addresses (though that scenario is more likely to be random in a real system). Then every structure would alias with the structure allocated 16 structures after it. Thus the likelihood of aliasing conflicts may increase with the sizes of the data structures.

Assembly/Compiler Coding Rule 28. (H impact, MH generality) *Lay out data or order computation to avoid having cache lines that have linear addresses that are a multiple of 64KB apart in the same working set. Avoid having more than 4 cache lines that are some multiple of 2KB apart in the same first-level cache working set, and avoid having more than 8 cache lines that are some multiple of 32KB apart in the same second-level cache working set. Avoid having a store followed by a non-dependent load with addresses that differ by a multiple of 16KB.*

When declaring multiple arrays that are referenced with the same index and are each a multiple of 64KB (as can happen with `struct_of_array` data layouts), pad them to avoid declaring them contiguously. Padding can be accomplished by either intervening declarations of other variables, or by artificially increasing the dimension.

User/Source Coding Rule 4. (H impact, ML generality) *Consider using a special memory allocation library to avoid aliasing.*

One way to implement a memory allocator to avoid aliasing is to allocate more than enough space and pad. For example, allocate structures that are 68KB instead of 64KB to avoid the 64KB aliasing, or have the allocator pad and return random offsets that are a multiple of 128 Bytes (the size of a cache line).

User/Source Coding Rule 5. (M impact, M generality) *When padding variable declarations to avoid aliasing, the greatest benefit comes from avoiding aliasing on second-level cache lines, suggesting an offset of 128 bytes or more.*

Mixing Code and Data

The Pentium 4 processor's aggressive prefetching and pre-decoding of instructions has two related effects:

- Self-modifying code works correctly, according to the Intel architecture processor requirements, but incurs a significant performance penalty. Avoid self-modifying code.

- Placing writable data in the code segment might be impossible to distinguish from self-modifying code. Writable data in the code segment might suffer the same performance penalty as self-modifying code.

Assembly/Compiler Coding Rule 29. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch.*

Tuning Suggestion 1. *Rarely, a performance problem may be noted due to executing data on a code page as instructions. The only condition where this is very likely to happen is following an indirect branch that is not resident in the trace cache. Only if a performance problem is clearly due to this problem, try moving the data elsewhere, or inserting an illegal opcode or a pause instruction immediately following the indirect branch. The latter two alternatives may degrade performance in some circumstances.*

Assembly/Compiler Coding Rule 30. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate pages, or at least in separate 1K regions.*

Write Combining

Write combining (WC) improves performance in two ways:

- On a write miss to the first-level cache, it allows multiple stores to the same cache line to occur before that cache line is read for ownership (RFO) from further out in the cache/memory hierarchy. Then the rest of line is read, and the bytes that have not been written to are combined with the unmodified bytes in the returned line.
- It allows multiple writes to be assembled and written further out in the cache hierarchy as a unit, saving port and bus traffic. This is particularly important for avoiding partial writes to uncached memory.

There are 6 write-combining buffers. Up to two of those buffers may be written out to higher cache levels and freed up for use on other write misses, so only four write-combining buffers are guaranteed to be available for simultaneous use.

Assembly/Compiler Coding Rule 31. (H impact, L generality) *If an inner loop writes to more than four arrays, apply loop fission to break up the body of the loop such that only four arrays are being written to in each iteration.*

The write combining buffers are used for stores of all memory types. They are particularly important for writes to uncached memory: writes to different parts of the same cache line can be grouped into a single, full-cache-line bus transaction instead of going across the bus (since they are not cached) as several partial writes. Avoiding partial writes can have a critical impact on bus bandwidth-bound graphics applications, where graphics buffers are in uncached memory. Separating writes to uncached memory and writes to writeback memory into separate phases can assure that the write combining buffers can fill before getting evicted by other write traffic. Eliminating partial write transactions has been found to have performance impact of the order of 20% for some applications. Because the cache lines are 64 bytes for Pentium 4 processor instead of 32 bytes for Pentium III processor, and the maximum bandwidth is higher for Pentium 4 processor, the potential gain for Pentium 4 processor is greater. See more on optimizations at <http://developer.intel.com/design/pentium4/manuals>.

Store ordering and visibility is another important issue for write combining. When a write to a write combining buffer occurs, there will be a read-for-ownership (RFO). If a subsequent write happens to another write combining buffer, a separate RFO will be caused for that cache line. The first cache line cannot be written again until the second RFO has been serviced to guarantee properly-ordered visibility of the writes, causing a delay. If the memory type for the writes is write combining, there will be no RFO since the line is not cached, and there is no such delay. For more details on write combining, see the *Intel Architecture Software Developer's Manual*.

Locality Enhancement

Although cache miss rates may be low, processors typically spend a sizable portion of their execution time waiting for cache misses to be serviced. Reducing cache misses by enhancing a program's locality is a key optimization. This can take several forms: blocking to iterate over a portion of an array that will fit in the cache, loop interchange to avoid crossing cache lines or page boundaries, and loop skewing to make accesses contiguous.

User/Source Coding Rule 6. (H impact, H generality) *Turn on loop optimizations in the compiler to enhance locality for nested loops.*

User/Source Coding Rule 7. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing and packing are best done by the compiler. Optimize data structures to either fit in one-half of the first-level cache or in the second-level cache.*

Optimizing for one-half of the first-level cache will bring the greatest performance benefit. If one-half of the first-level cache is too small to be practical, optimize for the second-level cache. Optimizing for a point in between (for example, for the entire first-level cache) will likely not bring a substantial improvement over optimizing for the second-level cache.

Prefetching

The Pentium 4 processor has three prefetching mechanisms:

- hardware instruction prefetcher
- software prefetch for data
- hardware prefetch for cache lines of data or instructions.

Hardware Instruction Fetching

The hardware instruction fetcher reads instructions, 32 bytes at a time, into the 64-byte instruction streaming buffers.

Software and Hardware Cache Line Fetching

The Pentium 4 processor provides hardware prefetching, in addition to software prefetching. The hardware prefetcher operates transparently to fetch data and instruction streams from memory, without requiring programmer's intervention.

Starting to prefetch data before it is actually needed for a load can reduce the wait time for the data and hence reduce the latency penalty of the load. The Pentium III and subsequent processors provide software prefetch instructions. The `prefetchnta` instruction is likely to be a good choice for most cases, because it brings the data close and doesn't pollute the caches.

Prefetching can provide significant gains, and the use of prefetches is recommended, particularly for regular strided accesses. It must be used carefully however, and there is a trade-off to be made between hardware and software prefetching, based on application characteristics such as regularity and stride of accesses, whether the problem is bus bandwidth, issue bandwidth or the latency of loads on the critical path, and whether the access patterns are suitable for non-temporal prefetch. An optimum implementation of software-controlled prefetch can be determined empirically.

For a detailed description of how to use prefetching, see [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#).

User/Source Coding Rule 8. (M impact, H generality) *Try using compiler-generated software prefetching if supported by the compiler you’re using. **Note:** As the compiler’s prefetch implementation improves, it is expected that its prefetch insertion will outperform manual insertion except for code tuning experts, but this is not always the case. If the compiler does not support software prefetching, intrinsics or inline assembly may be used to manually insert prefetch instructions.*

[Chapter 6](#) contains an example of using software prefetch to implement memory copy algorithm.

Tuning Suggestion 2. *If a load is found to miss frequently, either insert a prefetch before it, or, if issue bandwidth is a concern, move the load up to execute earlier.*

Cacheability instructions

SSE2 provides additional cacheability instructions that extend further from the cacheability instructions provided in SSE. The new cacheability instructions include:

- new streaming store instructions
- new cache line flush instruction
- new memory fencing instructions

For a detailed description of using cacheability instructions, see [Chapter 6](#).

Code

Because the trace cache (TC) removes the decoding stage from the pipeline for frequently executed code, optimizing code alignment for decoding is not as important a consideration as it was on prior generation processors.

Careful arrangement of code can enhance cache and memory locality. Likely sequences of basic blocks should be laid out contiguously in memory. This may involve pulling unlikely code, such as code to handle error conditions, out of that sequence. See “[Prefetching](#)” section on how to optimize for the instruction prefetcher.

Assembly/Compiler Coding Rule 32. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, the body of the conditional should be placed on a different code page.*

Improving the Performance of Floating-point Applications

When programming floating-point applications, it is best to start with a high-level programming language such as C, C++, or Fortran. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code, the compiler may need some assistance.

Guidelines for Optimizing Floating-point Code

User/Source Coding Rule 9. (M impact, M generality) *Target the Pentium 4 processor and enable the compiler's use of SSE2 instructions with appropriate switches.*

Follow this procedure to investigate the performance of your floating-point application:

- Understand how the compiler handles floating-point code.
- Look at the assembly dump and see what transforms are already performed on the program.
- Study the loop nests in the application that dominate the execution time.
- Determine why the compiler is not creating the fastest code.
- See if there is a dependence that can be resolved.
- Determine the problem area: bus bandwidth, cache locality, trace cache bandwidth, or instruction latency. Focus on optimizing the problem area. For example, adding prefetch instructions will not help if the bus is already saturated, and if trace cache bandwidth is the problem, the added prefetch μ ops may degrade performance.

For floating-point coding, follow all the general coding recommendations discussed throughout this chapter, including:

- blocking the cache
- using prefetch
- enabling vectorization
- unrolling loops.

User/Source Coding Rule 10. (H impact, ML generality) *Make sure your application stays in range.*

Out-of-range numbers cause very high overhead.

User/Source Coding Rule 11. (M impact, ML generality) *Do not use high precision unless necessary. Set the precision control (PC) field in the x87 FPU control word to "Single Precision". This allows single precision (32-bits) computation to complete faster on some operations (for example, divides). It also allows for an early out on divides. However, be careful of introducing more than a total of two values for the floating point control word, or there will be a large performance penalty. See ["Floating-point Modes"](#).*

User/Source Coding Rule 12. (H impact, ML generality) *Use fast float-to-int routines. If coding these routines, use the `cvtss2si`, `cvttsd2si` instructions if coding with Streaming SIMD Extensions 2.*

Many libraries do more work than is necessary. The instructions `cvtss2si`/`cvttsd2si` save many μ ops and some store-forwarding delays over some compiler implementations, and avoids changing the rounding mode.

User/Source Coding Rule 13. (M impact, ML generality) *Break dependence chains where possible.*

For example, to calculate $z = a + b + c + d$, instead of

```
x = a + b;
```

```
y = x + c;
```

```
z = y + d;
```

use

```
x = a + b;
```

```
y = c + d;
```

```
z = x + y;
```

User/Source Coding Rule 14. (M impact, ML generality) *Usually, math libraries take advantage of the transcendental instructions (for example, `fsin`) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider alternate, software-based approach, such as look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision, the size of the look-up table and taking advantage of the parallelism of the Streaming SIMD Extensions and the Streaming SIMD Extensions 2 instructions.*

Floating-point Modes and Exceptions

When working with floating-point numbers, high-speed microprocessors frequently must deal with situations that need special handling either by its hardware design or by coding techniques in software. The Pentium 4 processor is optimized to handle the most common cases of such situations efficiently.

Floating-point Exceptions

The most frequent situations that can lead to some performance degradations involve the masked floating-point exception conditions such as:

- arithmetic overflow
- arithmetic underflow
- denormalized operand

Refer to Chapter 4 of the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1 for the definition of overflow, underflow and denormal exceptions.

Denormalized floating-point numbers can impact performance in two ways:

- directly: when they are used as operands
- indirectly: when they are produced as a result of an underflow situation

If a floating-point application never underflows, the denormals can only come from floating-point constants.

User/Source Coding Rule 15. (H impact, ML generality) *Denormalized floating-point constants should be avoided as much as possible.*

Denormal and arithmetic underflow exceptions can occur during the execution of either x87 instructions or SSE/SSE2 instructions. The Pentium 4 processor can handle these exceptions more efficiently when executing SSE/SSE2 instructions and when speed is more important than complying to IEEE standard. The following two paragraphs give recommendations on how to optimize your code to reduce performance degradations related to floating-point exceptions.

Dealing with floating-point exceptions in x87 FPU code

Every instance of a special situation listed in [“Floating-point Exceptions”](#) is costly in terms of performance. For that reason, x87 FPU code should be written to avoid these special situations.

There are basically three ways to reduce the impact of overflow/underflow situations with x87 FPU code:

- Choose floating-point data types that are large enough to accommodate results without generating arithmetic overflow and underflow exceptions.
- Scale the range of operands/results to reduce as much as possible the number of arithmetic overflow/underflow situations
- Keep intermediate results on the x87 FPU register stack until the final results have been computed and stored to memory. Overflow or underflow is less likely to happen when intermediate results are kept in the x87 FPU stack (this is because data on the stack is stored in double extended-precision format and overflow/underflow conditions are detected accordingly).

Denormalized floating-point constants (which are read only, and hence never change) should be avoided and replaced, if possible, with zeros of the same sign.

Dealing with Floating-point Exceptions in SSE and SSE2 code

Most special situations that involve masked floating-point exception are handled very efficiently on the Pentium 4 processor. When masked overflow exception occurs while executing SSE or SSE2 code, the Pentium 4 processor handles this without performance penalty.

Underflow exceptions and denormalized source operands are usually treated according to the IEEE 754 specification. If a programmer is willing to trade pure IEEE 754 compliance for speed, two non-IEEE-754-compliant modes are provided to speed up situations where underflows and input are frequent: FTZ mode and DAZ mode.

When the FTZ mode is enabled, an underflow result is automatically converted to a zero with the correct sign. Although this behavior is not IEEE-754-compliant, it is provided to use in applications where performance is more important than pure IEEE-754 compliance. Since denormal results are not produced when the FTZ mode is enabled, the only denormal floating-point numbers that can be encountered are the ones that are constants (read only).

The DAZ mode is provided to handle denormal source operands efficiently when running an SSE application. When the DAZ mode is enabled, input denormals are treated as zeros with the same sign. Enabling the DAZ mode is the way to deal with denormal floating-point constants when performance is the objective.

If departing from IEEE 754 specification is acceptable, and especially if performance is critical, it is advised to run an SSE/SSE2 application with both FTZ and DAZ modes enabled.



NOTE. *The DAZ mode is available with both the SSE and SSE2 extensions, although the speed improvement expected from this mode is fully realized only in SSE code.*

Floating-point Modes

On the Pentium III processor, the `FLDCW` instruction is an expensive operation. On the Pentium 4 processor, the `FLDCW` instruction is improved for situations where an application alternates between two constant values of the x87 FPU control word (FCW), such as when performing conversions to integers.

Specifically, the optimization for the `FLDCW` instruction allows programmers to alternate between two constant values efficiently. For the `FLDCW` optimization to be effective, the two constant FCW values are only allowed to differ on the following 5 bits in the FCW:

FCW[8-9]	precision control
FCW[10-11]	rounding control
FCW[12]	infinity control.

If programmers need to modify other bits, for example, the mask bits, in the FCW, the `FLDCW` instruction is still an expensive operation.

In situations where an application cycles between three (or more) constant values, the `FLDCW` optimization does not apply and the performance degradation will occur for each `FLDCW` instruction.

One solution to this problem is to choose two constant FCW values, take advantage of the optimization of the `FLDCW` instruction to alternate between only these two constant FCW values, and devise some means to accomplish the task that required the 3rd FCW value without actually changing the FCW to a third constant value. An alternative solution is to structure the code, so that for some periods of time, the application first alternates between only two constant FCW values. When the application later alternates between a pair of different FCW values, the performance degradation occurs only briefly during the transition.

It is expected that SIMD applications are unlikely to alternate FTZ and DAZ mode values. Consequently, the SIMD control word does not have the short latencies that the floating-point control register does. A read of the `MXCSR` register has a fairly long latency, and a write is a serializing instruction.

There is no separate control word for single and double precision; both use the same modes. Notably, this applies to both FTZ and DAZ modes.

Assembly/Compiler Coding Rule 33. (H impact, M generality) *Minimize changes to bits 8-12 of the floating point control word. Changing among more than two values of these bits (precision, rounding and infinity control) leads to delays that are on the order of the pipeline depth.*

Rounding Mode

Many libraries provide the float-to-integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. With the Pentium 4 processor, one can use the `cvttsd2si` and `cvtss2si` instructions to convert operands with truncation and without ever needing to change rounding modes. The cost savings of using these instructions over the methods below is enough to justify using Streaming SIMD Extensions 2 and Streaming SIMD Extensions wherever possible when truncation is involved.

For x87 floating point, the `fist` instruction uses the rounding mode represented in the floating-point control word (FCW). The rounding mode is generally round to nearest, therefore many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the `fldcw` instruction. If this

is the only change in the rounding, precision, and infinity bits, then use the `fstcw` instruction to store the floating-point control word and then use the `fldcw` instruction to change the rounding mode to truncation.

In a typical code sequence that changes the rounding mode in the FCW, a `fstcw` instruction is usually followed by a load operation. The load operation from memory should be a 16-bit operand to prevent store-forwarding problem. If the load operation on the previously-stored FCW word involves either an 8-bit or a 32-bit operand, this will cause a store-forwarding problem due to mismatch of the size of the data between the store operation and the load operation.

Make sure that the write and read to the FCW are both 16-bit operations, to avoid store-forwarding problems.

Only if there is *more than one* change to the rounding, precision and infinity bits, and the rounding mode is not important to the results, then use the algorithm in Example 2-19 to avoid the synchronization and overhead of the `fldcw` instruction and changing the rounding mode. This example suffers from a store-forwarding problem which will lead to a severe performance penalty. However, its performance is still better than changing the rounding, precision and infinity bits among more than two values.

Example 2-19 Algorithm to Avoid Changing the Rounding Mode

```
_ftol32proc
    lea    ecx,[esp-8]
    sub    esp,16          ; allocate frame
    and    ecx,-8          ; align pointer on boundary of 8
    fld    st(0)           ; duplicate FPU stack top
    fistp  qword ptr[ecx]
    fild   qword ptr[ecx]
    mov    edx,[ecx+4]     ; high dword of integer
    mov    eax,[ecx]       ; low dword of integer
    test   eax,eax
    je     integer_QNaN_or_zero
```

continued

Example 2-19 Algorithm to Avoid Changing the Rounding Mode (continued)

```

arg is not integer QNaN:
    fsubp    st(1),st          ; TOS=d-round(d),
                                ; { st(1)=st(1)-st & pop ST}
    test     edx,edx          ; what's sign of integer
    jns      positive         ; number is negative
    fstp     dword ptr[ecx]    ; result of subtraction
    mov      ecx,[ecx]         ; dword of diff(single-
                                ; precision)

    add      esp,16
    xor      ecx,80000000h
    add      ecx,7fffffffh    ; if diff<0 then decrement
                                ; integer
    adc      eax,0             ; inc eax (add CARRY flag)
    ret

positive:
    fstp     dword ptr[ecx]    ; 17-18 result of subtraction
    mov      ecx,[ecx]         ; dword of diff(single precision)
    add      esp,16
    add      ecx,7fffffffh    ; if diff<0 then decrement integer
    sbb      eax,0             ; dec eax (subtract CARRY flag)
    ret

integer_QNaN_or_zero:
    test     edx,7fffffffh
    jnz      arg_is_not_integer_QNaN
    add      esp,16
    ret

```

Assembly/Compiler Coding Rule 34. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision and infinity bits.*

Precision

If single precision is adequate, it is recommended over double precision. This is true for two reasons:

- Single precision operations allow the use of longer SIMD vectors, since more single precision data elements fit in a register than double precision elements.
- If the precision control (PC) field in the x87 FPU control word is set to “Single Precision,” the floating-point divider can complete a single-precision computation much faster than either a double-precision computation or an extended double-precision computation. If the PC field is set to “Double Precision,” this will enable those x87 FPU operations on double-precision data to complete faster than extended double-precision computation. These characteristics affect computations including floating-point divide and square root.

Assembly/Compiler Coding Rule 35. (H impact, L generality) *Minimize the number of changes to the precision mode.*

Improving Parallelism and the Use of FXCH

The x87 instruction set relies on the floating point stack for one of its operands for most operations. If the dependence graph is a tree, which means each intermediate result is used only once, and code is scheduled carefully, it is often possible to use only operands that are on the top of the stack or in memory, and to avoid using operands that are buried under the top of the stack. When operands need to be pulled from the middle of the stack, an `fxch` instruction can be used to swap the operand on the top of the stack with another entry in the stack.

An `fxch` instruction can also be used to enhance parallelism. Dependent chains can be overlapped to expose more independent instructions to the hardware scheduler. An `fxch` instruction may be required to effectively increase the register name space so that more operands can be simultaneously live.

However, an `fxch` instruction inhibits issue bandwidth in the trace cache, not only because it consumes a slot, but also because of issue slot restrictions imposed on `fxch` instructions. If the application is not bound by issue or retirement bandwidth, the `fxch` instructions will have no impact.

The Pentium 4 processor's effective instruction window size is large enough to permit instructions that are as far away as the next iteration to be overlapped, often obviating the need for using `fxch` instructions to enhance parallelism.

Thus the `fxch` instruction should be used only when it is needed to express an algorithm, or to enhance parallelism where it can be shown to be lacking. If the size of the register name space is a problem, the use of the XMM registers is recommended, as described in the next section.

Assembly/Compiler Coding Rule 36. (M impact, M generality) *Use `fxch` only where necessary to increase the effective name space.*

This in turn allows instructions to be reordered to make instructions available to be executed in parallel. Out-of-order execution precludes the need for using `fxch` to move instructions for very short distances.

x87 vs. SIMD Floating-point Trade-offs

There are a number of differences between x87 floating-point code and scalar floating-point code using SSE and/or SSE2. These differences drive decisions about which registers and accompanying instructions to use:

- When an input operand for a SIMD floating-point instruction contains values that are less than the representable range of the data type, a denormal exception occurs, which causes significant performance penalty. SIMD floating-point operation has a flush-to-zero mode. In flush-to-zero mode, the results will not underflow. Therefore subsequent computation will not face the performance penalty of handling denormal input operands. For example, in a typical case of 3D applications with low lighting levels, using flush-to-zero mode can improve performance by as much as 50% on applications with a large number of underflows.
- Scalar floating point has lower latencies. This generally does not matter too much as long as resource utilization is low.
- Only x87 supports transcendental instructions.
- x87 supports 80-bit precision, double extended floating point. Streaming SIMD Extensions support a maximum of 32-bit precision, and Streaming SIMD Extensions 2 supports a maximum of 64-bit precision.

- On the Pentium 4 processor, floating point adds are pipelined for x87 but not for scalar floating-point code. Floating point multiplies are not pipelined for either case. Thus for applications with a large number of floating-point adds relative to the number of multiplies, x87 may be a better choice.
- The scalar floating-point registers may be accessed directly, avoiding `fxch` and top-of-stack restrictions. Furthermore, on the Pentium 4 processor, the floating-point register stack may be used simultaneously with the XMM registers. The same hardware is used for both kinds of instructions, but the added name space may be beneficial.
- The cost of converting from floating point to integer with truncation is significantly lower with Streaming SIMD Extensions 2 and Streaming SIMD Extensions in the Pentium 4 processor than with either changes to the rounding mode or the sequence prescribed in the [Example 2-19](#) above.

Assembly/Compiler Coding Rule 37. (M impact, M generality) *Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Use x87 floating-point adds if the ratio of floating-point adds to the number of floating-point multiplies is high.*

Memory Operands

Double-precision floating-point operands that are eight-byte aligned have better performance than operands that are not eight-byte aligned, since they are less likely to incur penalties for cache and MOB splits. Floating-point operation on a memory operands require that the operand be loaded from memory. This incurs an additional `μop`, which can have a minor negative impact on front end bandwidth. Additionally, memory operands may cause a data cache miss, causing a penalty.

Floating-Point Stalls

Floating-point instructions have a latency of at least two cycles. But, because of the out-of-order nature of Pentium II and the subsequent processors, stalls will not necessarily occur on an instruction or `μop` basis. However, if an instruction has a very long latency such as an `fdiv`, then scheduling can improve the throughput of the overall application.

x87 Floating-point Operations with Integer Operands

For Pentium 4 processor, splitting floating-point operations (`fiadd`, `fisub`, `fimul`, and `fidiv`) that take 16-bit integer operands into two instructions (`fild` and a floating-point operation) is more efficient. However, for floating-point operations with 32-bit integer operands, using `fiadd`, `fisub`, `fimul`, and `fidiv` is equally efficient compared with using separate instructions.

Assembly/Compiler Coding Rule 38. (M impact, L generality) *Try to use 32-bit operands rather than 16-bit operands for `fild`. However, do not do so at the expense of introducing a store forwarding problem by writing the two halves of the 32-bit memory operand separately.*

x87 Floating-point Comparison Instructions

On Pentium II and the subsequent processors, the `fcomi` and `fcmov` instructions should be used when performing floating-point comparisons. Using (`fcom`, `fcomp`, `fcompp`) instructions typically requires additional instruction like `fstsw`. The latter alternative causes more ops to be decoded, and should be avoided.

Transcendental Functions

If an application needs to emulate these math functions in software due to performance or other reasons (see “[Guidelines for Optimizing Floating-point Code](#)”), it may be worthwhile to inline some of these math library calls because the `call` and the prologue/epilogue involved with the calls can significantly affect the latency of the operations.

Note that transcendental functions are supported only in x87 floating point, not in Streaming SIMD Extensions or Streaming SIMD Extensions 2.

Instruction Selection

This section explains which instruction sequences to avoid or what alternative code sequences to use when generating optimal assembly code. These optimizations have been shown to contribute to the overall performance at the application level on the order of 5%, across many applications. Although performance gain for individual application will vary by benchmark.

The prioritized order of recommendations for instruction selection is:

- Choose instructions with shorter latencies and fewer μ ops.
- Use optimized sequences for clearing and comparing registers.
- Enhance register availability.
- Avoid prefixes, especially more than one prefix.

A compiler may be already doing a good job on instruction selection as it is. In that case, user intervention usually is not necessary.

Complex Instructions

Assembly/Compiler Coding Rule 39. (ML impact, M generality) *Avoid using complex instructions (for example, `enter`, `leave`, or `loop`) that generally have more than four μ ops and require multiple cycles to decode. Use sequences of simple instructions instead.*

Complex instructions may save architectural registers, but incur a penalty of 4 μ ops to set up parameters for the microcode ROM.

Use of the `lea` Instruction

In many cases an `lea` instruction or a sequence of `lea`, `add`, `sub`, and `shift` instructions can be used to replace constant multiply instructions. The `lea` instruction can be used sometimes as a three or four operand addition instruction, for example,

```
lea ecx, [eax + ebx + 4 + a]
```

Using the `lea` instruction in this way can avoid some unnecessary register usage by not tying up registers for the operands of some arithmetic instructions. It may also save code space.

The `lea` instruction is not always as fast on the Pentium 4 processor as it is on the Pentium II and Pentium III processors. This is primarily due to the fact that the `lea` instruction can produce a shift μ op. If the `lea` instruction uses a shift by a constant amount then the latency of the sequence of μ ops is shorter if `adds` are used instead of a shift, and the `lea` instruction is replaced with the appropriate sequence of μ ops. However, this increases the total number of μ ops, leading to a trade-off:

Assembly/Compiler Coding Rule 40. (ML impact, M generality) *If an `lea` instruction which uses the scaled index is on the critical path, the sequence with the `adds` may be better, but if code density and bandwidth out of the trace cache are the critical factor, then the `lea` instruction should be used.*

Use of the `inc` and `dec` Instructions

The `inc` and `dec` instructions modify only a subset of the bits in the flag register. This creates a dependence on all previous writes of the flag register. This is especially problematic when these instructions are on the critical path because they are used to change an address for a load on which many other instructions depend.

Assembly/Compiler Coding Rule 41. (M impact, H generality) *`inc` and `dec` instructions should be replaced with an `add` or `sub` instruction, because `add` and `sub` overwrite all flags.*

The optimization of implementing Coding Rule 41 benefits Pentium 4 and future IA-32 processors based on the Intel NetBurst micro-architecture, although it does not help Pentium II processors, and it adds an additional byte per instruction.

Use of the `shift` and `rotate` Instructions

The `shift` and `rotate` instructions have a longer latency on the Pentium 4 processor than on previous processor generations. The latency of a sequence of `adds` will be shorter for left shifts of three or less. Fixed and variable shifts have the same latency.

Assembly/Compiler Coding Rule 42. (M impact, M generality) *If a `shift` is on a critical path, replace it by a sequence of up to three `adds`. If its latency is not critical, use the `shift` instead because it produces fewer `μops`.*

The `rotate by immediate` and `rotate by register` instructions are more expensive than a `shift`. The `rotate by 1` instruction has the same latency as a `shift`.

Assembly/Compiler Coding Rule 43. (ML impact, L generality) *Avoid `rotate by register` or `rotate by immediate` instructions. If possible, replace with a `rotate by 1` instruction.*

Integer and Floating-point Multiply

The integer multiply operations, `mul` and `imul`, are executed in the floating-point unit so these instructions should not be executed in parallel with a floating-point instruction. They also incur some extra latency due to being executed on the floating-point unit.

A floating-point multiply instruction (`fmul`) delays for one cycle if the immediately preceding cycle executed an `fmul`. The multiplier can only accept a new pair of operands every other cycle.

Assembly/Compiler Coding Rule 44. (M impact, MH generality) *Replace integer multiplies by a small constant with two or more `add` and `lea` instructions, especially when these multiplications is part of a dependence chain.*

Integer Divide

Typically, an integer divide is preceded by a `cwd` or `cdq` instruction. Depending on the operand size, divide instructions use `DX:AX` or `EDX:EAX` as the dividend. The `cwd` or `cdq` instructions sign-extend `AX` or `EAX` into `DX` or `EDX`, respectively. These instructions are denser encoding than a `shift` and `move` would be, but they generate the same number of μ ops. If `AX` or `EAX` are known to be positive, replace these instructions with

```
xor dx, dx
```

or

```
xor edx, edx
```

Assembly/Compiler Coding Rule 45. (ML impact, L generality) *Use `cdw` or `cdq` instead of a `shift` and a `move`. Replace these with an `xor` whenever `AX` or `EAX` is known to be positive.*

Operand Sizes

The Pentium 4 processor does not incur a penalty for partial register accesses as did the Pentium II and Pentium III processors, since every operation on a partial register updates the whole register. However, this does mean that there may be false dependencies between *any* references to partial registers. [Example 2-20](#) demonstrates a series of false dependencies caused by referencing partial registers.

Example 2-20 False Dependencies Caused by Referencing Partial Registers

```
1:  add    ah, bh
2:  add    al, 3      ; instructions 2 has a false dependency on 1
3:  mov    bl, al
4:  mov    ax, cx      ; instructions 4 has a false dependency on 2
5:  imul   eax, 3      ; instructions 5 has a false dependency on 4
6:  mov    al, bl      ; instructions 6 has a false dependency on 5
7:  add    al, 13      ; instructions 7 has a false dependency on 6
8:  imul   dl, al      ; instructions 8 has a false dependency on 7
9:  mov    al, 17      ; instructions 9 has a false dependency on 7
```

If instructions 4, 6 and 9 in Example 2-20 are changed to use a `movzx` instruction instead of a `mov`, then the dependences of instructions 4 on 2 (and transitively 1 before it), instructions 6 on 5 and instructions 9 on 7 are broken, creating three independent chains of computation instead of one serial one. Especially in a tight loop with limited parallelism, this optimization can yield several percent performance improvement.

Assembly/Compiler Coding Rule 46. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using `movzx`.*

On Pentium II processors, the `movsx` and `movzx` instructions both take a single μop , whether they move from a register or memory. On Pentium 4 processors, the `movsx` takes an additional μop . This is likely to cause less delay than the partial register update problem above, but the performance gain may vary. If the additional μop is a critical problem, `movsx` can sometimes be used as alternative. For example, sometimes sign-extended semantics can be maintained by zero-extending operands. For example, the C code in the following statements does not need sign extension, nor does it need prefixes for operand size overrides:

```
static short int a, b;
if (a==b) {
    . . .
}
```

Code for comparing these 16-bit operands might be:

```
movzw  eax, [a]
movzw  ebx, [b]
cmp     eax, ebx
```

The circumstances, when this technique can be applicable, tend to be quite common. However, this technique would not work if the compare was for greater than, less than, greater than or equal, and so on, or if the values in `eax` or `ebx` were to be used in another operation where sign extension was required.

Assembly/Compiler Coding Rule 47. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.*

The trace cache can be packed more tightly when instructions with operands that can only be represented as 32 bits are not adjacent.

Assembly/Compiler Coding Rule 48. (ML impact, M generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as a sign-extended 16-bit immediate near each other. Try to schedule μ ops that have no immediate immediately before or after μ ops with 32-bit immediates.*

Address Calculations

Use the addressing modes for computing addresses rather than using the general-purpose computation. Internally, memory reference instructions can have four operands:

- relocatable load-time constant
- immediate constant
- base register
- scaled index register.

In the segmented model, a segment register may constitute an additional operand in the linear address calculation. In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

Clearing Registers

Pentium 4 processor provides special support to `xor`, `sub`, or `pxor` operations, specifically when executed within the same register, recognizing that clearing a register does not depend on the old value of the register. The `xorps` and `xorpd` instructions do not have this special support, and cannot be used to break dependence chains.

Assembly/Compiler Coding Rule 49. (M impact, ML generality) Use `xor`, `sub`, or `pxor` to set a register to 0, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move 0 into the register instead. This requires more code space than using `xor` and `sub`, but avoids setting the condition codes.

Compares

Use `test` when comparing a value in a register with zero. `Test` essentially `ands` the operands together without writing to a destination register. `Test` is preferred over `and` because `and` produces an extra result register. `Test` is better than `cmp ... , 0` because the instruction size is smaller.

Use `test` when comparing the result of a logical `and` with an immediate constant for equality or inequality if the register is `eax` for cases such as:

```
if (avar & 8) { }
```

The `test` instruction can also be used to detect rollover of modulo a power of 2. For example, the C code:

```
if ( (avar % 16) == 0 ) { }
```

can be implemented using:

```
test eax, 0x0F
jnz AfterIf
```

Assembly/Compiler Coding Rule 50. (ML impact, M generality) Use the `test` instruction instead of `and` or `cmp` if possible.

Often a produced value must be compared with zero, and then used in a branch. Because most Intel architecture instructions set the condition codes as part of their execution, the compare instruction may be eliminated. Thus the operation can be tested directly by a `jcc` instruction. The most notable exceptions are `mov` and `leaq`. In these cases, use `test`.

Assembly/Compiler Coding Rule 51. (ML impact, M generality) *Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a `test` instruction instead of a `compare`. Be certain that any code transformations made do not introduce problems with overflow.*

Floating Point/SIMD Operands

Beware that in the initial Pentium 4 processor implementation, the latency of MMX or SIMD floating point register to register moves is quite long. This may have implications for register allocation. However, this characteristic is not inherent to the operation, and this latency could change significantly on future implementations.

Moves that write only a portion of a register can introduce unwanted dependences. The `movsd reg, reg` instruction writes only the bottom 64 bits of a register, not all 128 bits. This introduces a dependence on the preceding instruction that produces the upper 64 bits, even if those bits are not longer wanted. The dependence inhibits the machine's register renaming, and hence reduces parallelism. An alternative is to use the `movapd` instruction, which writes all 128 bits. Even though the latter has a longer latency, the μ ops for `movapd` use a different execution port, which is more likely to be free. This change can have a several percent impact on performance. There may be exceptional cases where the latency matters more than the dependence or the execution port.

Assembly/Compiler Coding Rule 52. (M impact, ML generality) *Avoid introducing dependences with partial floating point register writes, e.g. from the `movsd xmmreg1, xmmreg2` instruction. Use the `movapd xmmreg1, xmmreg2` instruction instead.*

The `movsd xmmreg, mem`, however, writes all 128 bits, and hence breaks a dependence.

The `movupd` from memory instruction performs two 64-bit loads, but requires additional `µops` to adjust the address and combine the loads into a single register. This same functionality can be obtained using `movsd xmmreg1, mem; movsd xmmreg2, mem+8; unpcklpd xmmreg1, xmmreg2`, which uses fewer `µops` and can be packed into the trace cache more effectively. The latter alternative has been found to provide several percent of performance improvement in some cases. Its encoding requires more instruction bytes, but this is seldom an issue for the Pentium 4 processor. The store version of `movupd` is complex and slow, such that the sequence with two `movsd` and a `unpckhpd` should always be used.

Assembly/Compiler Coding Rule 53. (ML impact, L generality) *Instead of using `movupd xmmreg1, mem` for a unaligned 128-bit load, use `movsd xmmreg1, mem; movsd xmmreg2, mem+8; unpcklpd xmmreg1, xmmreg2`. If the additional register is not available, then use `movsd xmmreg1, mem; movhpd xmmreg1, mem+8`.*

Assembly/Compiler Coding Rule 54. (M impact, ML generality) *Instead of using `movupd mem, xmmreg1` for a store, use `movsd mem, xmmreg1; unpckhpd xmmreg1, xmmreg1; movsd mem+8, xmmreg1` instead.*

Prolog Sequences

Assembly/Compiler Coding Rule 55. (M impact, MH generality) *In routines that do not need `EBP` and that do not have called routines that modify `ESP`, use `ESP` as the base register to free up `EBP`. This optimization does not apply in the following cases: a routine is called that leaves `ESP` modified upon return, for example, `alloca`; routines that rely on `EBP` for structured or C++-style exception handling; routines that use `setjmp` and `longjmp`; and routines that rely on `EBP` debugging.*

If you are not using the 32-bit flat model, remember that `EBP` cannot be used as a general purpose base register because it references the stack segment.

Code Sequences that Operate on Memory Operands

Careful management of memory operands can improve performance. Instructions of the form “`OP REG, MEM`” can reduce register pressure by taking advantage of hidden scratch registers that are not available to the compiler.

Assembly/Compiler Coding Rule 56. (M impact, ML generality) *Instead of explicitly loading the memory operand into a register and then operating on it, reduce register pressure by using the memory operand directly, if that memory operand is not reused soon.*

The recommended strategy is as follows:

1. Initially, operate on register operands and use explicit load and store instructions, minimizing the number of memory accesses by merging redundant loads.
2. In a subsequent pass, free up the registers that contain the operands that were in memory for other uses by replacing any detected code sequence of the form shown in [Example 2-21](#) with `OP REG2, MEM1`.

Example 2-21 Recombining LOAD/OP Code into REG, MEM Form

```
LOAD reg1, mem1
... code that does not write to reg1...
OP   reg2, reg1
... code that does not use reg1 ...
```

Using memory as a destination operand may further reduce register pressure at the slight risk of making trace cache packing more difficult.

On the Pentium 4 processor, the sequence of loading a value from memory into a register and adding the results in a register to memory is faster than the alternate sequence of adding a value from memory to a register and storing the results in a register to memory. The first sequence also uses one less μ op than the latter.

Assembly/Compiler Coding Rule 57. (ML impact, M generality) *Give preference to adding a register to memory (memory is the destination) instead of adding memory to a register.*

Instruction Scheduling

Ideally, scheduling or pipelining should be done in a way that optimizes performance across all processor generations. This section presents scheduling rules that can improve the performance of your code on the Pentium 4 processor.

Latencies and Resource Constraints

Assembly/Compiler Coding Rule 58. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads.*

Spill Scheduling

The spill scheduling algorithm used by a code generator will be impacted by the Pentium 4 processor memory subsystem. A spill scheduling algorithm is an algorithm that selects what values to spill to memory when there are too many live values to fit in registers. Consider the code in [Example 2-22](#), where it is necessary to spill either A, B, or C.

Example 2-22 Spill Scheduling Example Code

```
LOOP
  C := ...
  B := ...
  A := A + ...
```

For the Pentium 4 processor, using dependence depth information in spill scheduling is even more important than in previous processors. The loop- carried dependence in A makes it especially important that A not be spilled. Not only would a store/load be placed in the dependence chain, but there would also be a data-not-ready stall of the load, costing further cycles.

Assembly/Compiler Coding Rule 59. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies.*

A possibly counter-intuitive result: in such a situation it is better to put loop invariants in memory than in registers, since loop invariants never have a load blocked by store data that is not ready.

Scheduling Rules for the Pentium 4 Processor Decoder

The Pentium 4 processor has a single decoder that can decode instructions at the maximum rate of one instruction per clock. Complex instruction must enlist the help of the microcode ROM; see [Chapter 1, “Intel® Pentium® 4 Processor Overview”](#), for more details.

Unlike the Pentium II and Pentium III processors, there is no need to schedule for decoders with different capabilities.

Vectorization

This section provides a brief summary of optimization issues related to vectorization. Chapters 3, 4 and 5 provide greater detail.

Vectorization is a program transformation which allows special hardware to perform the same operation of multiple data elements at the same time. Successive processor generations have provided vector support through the MMX technology, Streaming SIMD Extensions technology and Streaming SIMD Extensions 2. Vectorization is a special case of SIMD, a term defined in Flynn’s architecture taxonomy to denote a Single Instruction stream capable of operating on Multiple Data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating point data elements in Streaming SIMD Extensions and two double-precision floating- point data elements in Streaming SIMD Extensions 2 to sixteen byte operations in a 128-bit register in Streaming SIMD Extensions 2. Thus the vector length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

The Intel C++ Compiler supports vectorization in three ways:

- The compiler may be able to generate SIMD code without intervention from the user.
- The user inserts pragmas to help the compiler realize that it can vectorize the code.
- The user may write SIMD code explicitly using intrinsics and C++ classes.

To help enable the compiler to generate SIMD code

- avoid global pointers
- avoid global variables

These may be less of a problem if all modules are compiled simultaneously, and whole-program optimization is used.

User/Source Coding Rule 16. (H impact, M generality) *Use the smallest possible data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible.*

User/Source Coding Rule 17. (M impact, ML generality) *Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence.*

The integer part of the SIMD instruction set extensions are primarily targeted for 16-bit operands. Not all of the operators are supported for 32 bits, meaning that some source code will not be able to be vectorized at all unless smaller operands are used.

User/Source Coding Rule 18. (M impact, ML generality) *Avoid the use of conditional branches inside loops.*

User/Source Coding Rule 19. (M impact, ML generality) *Keep induction (loop) variables expressions simple.*

Miscellaneous

This section explains separate guidelines that do not belong to any category described above.

NOPs

Code generators generate a no-operation (NOP) to align instructions. The NOPs are recommended for the following operations:

- 1-byte: `xchg EAX, EAX`
- 2-byte: `mov reg, reg`
- 3-byte: `lea reg, 0 (reg)` (8-bit displacement)
- 6-byte: `lea reg, 0 (reg)` (32-bit displacement)

These are all true NOPs, having no effect on the state of the machine except to advance the EIP. Because NOPs require hardware resources to decode and execute, use the least number of NOPs to achieve the desired padding.

The one byte NOP, `xchg EAX,EAX`, has special hardware support. Although it still consumes a μop and its accompanying resources, the dependence upon the old value of `EAX` is removed. Therefore, this μop can be executed at the earliest possible opportunity, reducing the number of outstanding instructions. This is the lowest cost NOP possible.

The other NOPs have no special hardware support. Their input and output registers are interpreted by the hardware. Therefore, a code generator should arrange to use the register containing the oldest value as input, so that the NOP will dispatch and release RS resources at the earliest possible opportunity.

Try to observe the following NOP generation priority:

- Select the smallest number of NOPs and pseudo-NOPs to provide the desired padding.
- Select NOPs that are least likely to execute on slower execution unit clusters.
- Select the register arguments of NOPs to reduce dependencies.

Summary of Rules and Suggestions

To summarize the rules and suggestions specified in this chapter, be reminded that coding recommendations are ranked in importance according to these two criteria:

- Local impact (referred to earlier as “impact”) – the difference that a recommendation makes to performance for a given instance.
- Generality – how frequently such instances occur across all application domains.

Again, understand that this ranking is intentionally very approximate, and can vary depending on coding style, application domain, and other factors. Throughout the chapter you observed references to these criteria using the high, medium and low priorities for each recommendation. In places where there was no priority assigned, the local impact or generality has been determined not to be applicable.

The sections that follow summarize the sets of rules and tuning suggestions referenced in the manual.

User/Source Coding Rules

User/Source Coding Rule 1. (L impact, L generality) *If some targets of an indirect branch are very predictable, correlate either with preceding branches or with the same branch, then convert the indirect branch into a tree where one or more indirect branches are preceded by conditional branches to those targets* 2-20

User/Source Coding Rule 2. (H impact, M generality) *Pad data structures defined in the source code so that every data element is aligned to a natural operand size address boundary* 2-31

User/Source Coding Rule 3. (M impact, L generality) *Beware of false sharing within 64-byte cache lines* 2-34

User/Source Coding Rule 4. (H impact, ML generality) *Consider using a special memory allocation library to avoid aliasing.* 2-36

User/Source Coding Rule 5. (M impact, M generality) *When padding variable declarations to avoid aliasing, the greatest benefit comes from avoiding aliasing on second-level cache lines, suggesting an offset of 128 bytes or more.* 2-36

User/Source Coding Rule 6. (H impact, H generality) *Turn on loop optimizations in the compiler to enhance locality for nested loops.* 2-38

User/Source Coding Rule 7. (H impact, H generality) *Optimization techniques such as blocking, loop interchange, loop skewing and packing are best done by the compiler. Optimize data structures to either fit in one-half of the first-level cache or in the second-level cache.* 2-38

User/Source Coding Rule 8. (M impact, H generality) *Try using compiler-generated software prefetching if supported by the compiler you're using. Note: As the compiler's prefetch implementation improves, it is expected that its prefetch insertion will outperform manual insertion except for code tuning experts, but this is not always the case. If the compiler does not support software prefetching, intrinsics or inline assembly may be used to manually insert prefetch instructions.* 2-40

User/Source Coding Rule 9. (M impact, M generality) *Target the Pentium 4 processor and enable the compiler's use of SSE2 instructions with appropriate switches.* 2-41

User/Source Coding Rule 10. (H impact, ML generality) *Make sure your application stays in range* 2-42

User/Source Coding Rule 11. (M impact, ML generality) *Do not use high precision unless necessary. Set the precision control (PC) field in the x87 FPU control word to "Single Precision". This allows single precision (32-bits) computation to complete faster on some*

operations (for example, divides). It also allows for an early out on divides. However, be careful of introducing more than a total of two values for the floating point control word, or there will be a large performance penalty. See [“Floating-point Modes”](#). 2-42

User/Source Coding Rule 12. (H impact, ML generality) Use fast float-to-int routines. If coding these routines, use the `cvttss2si`, `cvttss2si` instructions if coding with Streaming SIMD Extensions 2. 2-42

User/Source Coding Rule 13. (M impact, ML generality) Break dependence chains where possible. 2-42

User/Source Coding Rule 14. (M impact, ML generality) Usually, math libraries take advantage of the transcendental instructions (for example, `fsin`) when evaluating elementary functions. If there is no critical need to evaluate the transcendental functions using the extended precision of 80 bits, applications should consider alternate, software-based approach, such as look-up-table-based algorithm using interpolation techniques. It is possible to improve transcendental performance with these techniques by choosing the desired numeric precision, the size of the look-up table and taking advantage of the parallelism of the Streaming SIMD Extensions and the Streaming SIMD Extensions 2 instructions. 2-42

User/Source Coding Rule 15. (H impact, ML generality) Denormalized floating-point constants should be avoided as much as possible. 2-43

User/Source Coding Rule 16. (H impact, M generality) Use the smallest possible data type, to enable more parallelism with the use of a (longer) SIMD vector. For example, use single precision instead of double precision where possible. 2-64

User/Source Coding Rule 17. (M impact, ML generality) Arrange the nesting of loops so that the innermost nesting level is free of inter-iteration dependencies. Especially avoid the case where the store of data in an earlier iteration happens lexically after the load of that data in a future iteration, something which is called a lexically backward dependence. 2-64

User/Source Coding Rule 18. (M impact, ML generality) Avoid the use of conditional branches inside loops. 2-64

User/Source Coding Rule 19. (M impact, ML generality) Keep induction (loop) variables expressions simple. 2-64

Assembly/Compiler Coding Rules

Assembly/Compiler Coding Rule 1. (MH impact, H generality) Arrange code to make basic blocks contiguous to eliminate unnecessary branches. 2-12

Assembly/Compiler Coding Rule 2. (M impact, ML generality) Use the `setcc` and `cmov` instructions to eliminate unpredictable conditional branches where possible. Do not do this for predictable branches. Also, do not use these instructions to eliminate all unpredictable conditional branches. Because using these instructions will incur execution overhead due to executing both paths of a conditional branch; Use these instructions only if the increase in computation time is less than the expected cost of a mispredicted branch. 2-13

Assembly/Compiler Coding Rule 3. (M impact, H generality) Arrange code to be consistent with the static branch prediction algorithm: make the fall-through code following a conditional branch be the likely target for a branch with a forward target, and make the fall-through code following a conditional branch be the unlikely target for a branch with a backward target. 2-16

Assembly/Compiler Coding Rule 4. (L impact, MH generality) Do not use directional branch hints if it is possible to position code to be consistent with the static branch prediction algorithm 2-18.

Assembly/Compiler Coding Rule 5. Use directional branch hints only in the case if the probability of the branch being taken in the prescribed direction is greater than 50%. Use code positioning to adhere to the static prediction algorithm wherever possible. 2-18

Assembly/Compiler Coding Rule 6. (MH impact, MH generality) Near calls must be matched with near returns, and far calls must be matched with far returns. Pushing the return address on the stack and jumping to the routine to be called is not recommended since it creates a mismatch in calls and returns. 2-19

Assembly/Compiler Coding Rule 7. (MH impact, MH generality) Selectively inline a function where doing so decreases code size, or if the function is small and the call site is frequently executed. 2-19

Assembly/Compiler Coding Rule 8. (H impact, M generality) Do not inline a function if doing so increases the working set size beyond what will fit in the trace cache. 2-19

Assembly/Compiler Coding Rule 9. (ML impact, ML generality) If there are more than 16 nested calls and returns in rapid succession, then consider transforming the program, for example, with `inline`, to reduce the call depth. 2-19

Assembly/Compiler Coding Rule 10. (ML impact, ML generality) *Favor inlining small functions that contain branches with poor prediction rates. If a branch misprediction results in a RETURN being prematurely predicted as taken, a performance penalty may be incurred. 2-19*

Assembly/Compiler Coding Rule 11. (M impact, MH generality) *If the average number of total iterations is less than or equal to 100, use a forward branch to exit the loop. 2-19*

Assembly/Compiler Coding Rule 12. (H impact, M generality) *Unroll small loops until the overhead of the branch and the induction variable accounts, generally, for less than about 10% of the execution time of the loop. 2-20*

Assembly/Compiler Coding Rule 13. (H impact, M generality) *Avoid unrolling loops excessively, as this may thrash the TC. 2-21*

Assembly/Compiler Coding Rule 14. (M impact, M generality) *Unroll loops that are frequently executed and that have a predictable number of iterations to reduce the number of iterations to 16 or fewer, unless this increases code size so that the working set no longer fits in the trace cache. If the loop body contains more than one conditional branch, then unroll so that the number of iterations is 16/(# conditional branches). 2-21*

Assembly/Compiler Coding Rule 15. (H impact, H generality) *Align data on natural operand size address boundaries 2-23*

Assembly/Compiler Coding Rule 16. (H impact, H generality) *Promote variables to registers where profitable. 2-25*

Assembly/Compiler Coding Rule 17. (MH impact, H generality) *Eliminate redundant loads. 2-25*

Assembly/Compiler Coding Rule 18. (H impact, M generality) *Pass parameters in registers instead of on the stack where possible. 2-25*

Assembly/Compiler Coding Rule 19. (H impact, M generality) *A load that forwards from a store must have the same address start point and therefore the same alignment as the store data. 2-27*

Assembly/Compiler Coding Rule 20. (H impact, M generality) *The data of a load which is forwarded from a store must be completely contained within the store data. 2-27*

Assembly/Compiler Coding Rule 21. (H impact, ML generality) *If it is necessary to extract a non-aligned portion of stored data, read out the smallest aligned portion that completely contains the data and shift/mask the data as necessary. The penalty for not doing this is much higher than the cost of the shifts. 2-27*

Assembly/Compiler Coding Rule 22. (MH impact, ML generality) *Avoid several small loads after large stores to the same area of memory by using a single large read and register copies as needed. 2-27*

Assembly/Compiler Coding Rule 23. (H impact, M generality) *Space out loads from the store that forwards data to them. Note that placing intervening instructions between the load and store does not guarantee separation in time. 2-30*

Assembly/Compiler Coding Rule 24. (ML impact, M generality) *If a routine is small, space apart the loads and stores that manage registers temporarily stored on the stack by re-loading the registers in the same order that they were stored; that is, replace pushes and pops with loads and stores, with the stores in the reverse order of pops. 2-30*

Assembly/Compiler Coding Rule 25. (H impact, MH generality) *Where it is possible to do so without incurring other penalties, prioritize the allocation of variables to registers, as in register allocation and for parameter passing, so as to minimize the likelihood and impact of store-forwarding problems. Try not to store-forward data generated from a long latency instruction, e.g. mul, div. Avoid store-forwarding data for variables with the shortest store-load distance. Avoid store-forwarding data for variables with many and/or long dependence chains, and especially avoid including a store forward on a loop-carried dependence chain. 2-30*

Assembly/Compiler Coding Rule 26. (H impact, M generality) *Try to arrange data structures such that they permit sequential access. 2-33*

Assembly/Compiler Coding Rule 27. (H impact, M generality) *If 64-bit data is ever passed as a parameter or allocated on the stack, make sure that the stack is aligned to an 8-byte boundary. 2-34*

Assembly/Compiler Coding Rule 28. (H impact, MH generality) *Lay out data or order computation to avoid having cache lines that have linear addresses that are a multiple of 64KB apart in the same working set. Avoid having more than 4 cache lines that are some multiple of 2KB apart in the same first-level cache working set, and avoid having more than 8 cache lines that are some multiple of 32KB apart in the same second-level cache working set. Avoid having a store followed by a non-dependent load with addresses that differ by a multiple of 16KB. 2-36*

Assembly/Compiler Coding Rule 29. (M impact, L generality) *If (hopefully read-only) data must occur on the same page as code, avoid placing it immediately after an indirect jump. For example, follow an indirect jump with its mostly likely target, and place the data after an unconditional branch. 2-37*

Assembly/Compiler Coding Rule 30. (H impact, L generality) *Always put code and data on separate pages. Avoid self-modifying code wherever possible. If code is to be modified, try to do it all at once and make sure the code that performs the modifications and the code being modified are on separate pages, or at least in separate 1K regions.* 2-37

Assembly/Compiler Coding Rule 31. (H impact, L generality) *If an inner loop writes to more than four arrays, apply loop fission to break up the body of the loop such that only four arrays are being written to in each iteration.* 2-37

Assembly/Compiler Coding Rule 32. (M impact, H generality) *If the body of a conditional is not likely to be executed, it should be placed in another part of the program. If it is highly unlikely to be executed and code locality is an issue, the body of the conditional should be placed on a different code page.* 2-41

Assembly/Compiler Coding Rule 33. (H impact, M generality) *Minimize changes to bits 8-12 of the floating point control word. Changing among more than two values of these bits (precision, rounding and infinity control) leads to delays that are on the order of the pipeline depth.* 2-46

Assembly/Compiler Coding Rule 34. (H impact, L generality) *Minimize the number of changes to the rounding mode. Do not use changes in the rounding mode to implement the floor and ceiling functions if this involves a total of more than two values of the set of rounding, precision and infinity bits.* 2-48

Assembly/Compiler Coding Rule 35. (H impact, L generality) *Minimize the number of changes to the precision mode.* 2-49

Assembly/Compiler Coding Rule 36. (M impact, M generality) *Use `fxch` only where necessary to increase the effective name space.* 2-50

Assembly/Compiler Coding Rule 37. (M impact, M generality) *Use Streaming SIMD Extensions 2 or Streaming SIMD Extensions unless you need an x87 feature. Use x87 floating-point adds if the ratio of floating-point adds to the number of floating-point multiplies is high.* 2-51

Assembly/Compiler Coding Rule 38. (M impact, L generality) *Try to use 32-bit operands rather than 16-bit operands for `fild`. However, do not do so at the expense of introducing a store forwarding problem by writing the two halves of the 32-bit memory operand separately.* 2-52

Assembly/Compiler Coding Rule 39. (ML impact, M generality) *Avoid using complex instructions (for example, `enter`, `leave`, or `loop`) that generally have more than four μ ops and require multiple cycles to decode. Use sequences of simple instructions instead.* 2-53

Assembly/Compiler Coding Rule 40. (ML impact, M generality) *If an `lea` instruction which uses the scaled index is on the critical path, the sequence with the `adds` may be better, but if code density and bandwidth out of the trace cache are the critical factor, then the `lea` instruction should be used.* 2-54

Assembly/Compiler Coding Rule 41. (M impact, H generality) *`inc` and `dec` instructions should be replaced with an `add` or `sub` instruction, because `add` and `sub` overwrite all flags.* 2-54

Assembly/Compiler Coding Rule 42. (M impact, M generality) *If a `shift` is on a critical path, replace it by a sequence of up to three `adds`. If its latency is not critical, use the `shift` instead because it produces fewer μ ops.* 2-54

Assembly/Compiler Coding Rule 43. (ML impact, L generality) *Avoid `rotate by register` or `rotate by immediate` instructions. If possible, replace with a `rotate by 1` instruction.* 2-54

Assembly/Compiler Coding Rule 44. (M impact, MH generality) *Replace integer multiplies by a small constant with two or more `add` and `lea` instructions, especially when these multiplications is part of a dependence chain.* 2-55

Assembly/Compiler Coding Rule 45. (ML impact, L generality) *Use `cdw` or `cdq` instead of a `shift` and a `move`. Replace these with an `xor` whenever `AX` or `EAX` is known to be positive.* 2-55

Assembly/Compiler Coding Rule 46. (M impact, MH generality) *Break dependences on portions of registers between instructions by operating on 32-bit registers instead of partial registers. For moves, this can be accomplished with 32-bit moves or by using `movzx`.* 2-56

Assembly/Compiler Coding Rule 47. (M impact, M generality) *Try to use zero extension or operate on 32-bit operands instead of using moves with sign extension.* 2-57

Assembly/Compiler Coding Rule 48. (ML impact, M generality) *Avoid placing instructions that use 32-bit immediates which cannot be encoded as a sign-extended 16-bit immediate near each other. Try to schedule μ ops that have no immediate immediately before or after μ ops with 32-bit immediates.* 2-57

Assembly/Compiler Coding Rule 49. (M impact, ML generality) Use `xor`, `sub`, or `pxor` to set a register to 0, or to break a false dependence chain resulting from re-use of registers. In contexts where the condition codes must be preserved, move 0 into the register instead. This requires more code space than using `xor` and `sub`, but avoids setting the condition codes. 2-58

Assembly/Compiler Coding Rule 50. (ML impact, M generality) Use the `test` instruction instead of `and` or `cmp` if possible. 2-58

Assembly/Compiler Coding Rule 51. (ML impact, M generality) Eliminate unnecessary compare with zero instructions by using the appropriate conditional jump instruction when the flags are already set by a preceding arithmetic instruction. If necessary, use a `test` instruction instead of a `compare`. Be certain that any code transformations made do not introduce problems with overflow. 2-59

Assembly/Compiler Coding Rule 52. (M impact, ML generality) Avoid introducing dependences with partial floating point register writes, e.g. from the `movsd xmmreg1, xmmreg2` instruction. Use the `movapd xmmreg1, xmmreg2` instruction instead. 2-59

Assembly/Compiler Coding Rule 53. (ML impact, L generality) Instead of using `movupd xmmreg1, mem` for a unaligned 128-bit load, use `movsd xmmreg1, mem; movsd xmmreg2, mem+8; unpcklpd xmmreg1, xmmreg2`. If the additional register is not available, then use `movsd xmmreg1, mem; movhpd xmmreg1, mem+8`. 2-60

Assembly/Compiler Coding Rule 54. (M impact, ML generality) Instead of using `movupd mem, xmmreg1` for a store, use `movsd mem, xmmreg1; unpckhpd xmmreg1, xmmreg1; movsd mem+8, xmmreg1` instead. 2-60

Assembly/Compiler Coding Rule 55. (M impact, MH generality) In routines that do not need `EBP` and that do not have called routines that modify `ESP`, use `ESP` as the base register to free up `EBP`. This optimization does not apply in the following cases: a routine is called that leaves `ESP` modified upon return, for example, `alloca`; routines that rely on `EBP` for structured or C++-style exception handling; routines that use `set jmp` and `long jmp`; and routines that rely on `EBP` debugging. 2-60

Assembly/Compiler Coding Rule 56. (M impact, ML generality) Instead of explicitly loading the memory operand into a register and then operating on it, reduce register pressure by using the memory operand directly, if that memory operand is not reused soon. 2-61

Assembly/Compiler Coding Rule 57. (ML impact, M generality) Give preference to adding a register to memory (memory is the destination) instead of adding memory to a register. 2-61

Assembly/Compiler Coding Rule 58. (M impact, MH generality) *Calculate store addresses as early as possible to avoid having stores block loads. 2-62*

Assembly/Compiler Coding Rule 59. (H impact, MH generality) *For small loops, placing loop invariants in memory is better than spilling loop-carried dependencies. 2-62*

Tuning Suggestions

Tuning Suggestion 1. *Rarely, a performance problem may be noted due to executing data on a code page as instructions. The only condition where this is very likely to happen is following an indirect branch that is not resident in the trace cache. Only if a performance problem is clearly due to this problem, try moving the data elsewhere, or inserting an illegal opcode or a pause instruction immediately following the indirect branch. The latter two alternative may degrade performance in some circumstances. 2-37*

Tuning Suggestion 2. *If a load is found to miss frequently, either insert a prefetch before it, or, if issue bandwidth is a concern, move the load up to execute earlier. 2-40*

Coding for SIMD Architectures

3

The Intel Pentium 4 processor includes support for Streaming SIMD Extensions 2, Streaming SIMD Extensions technology, and MMX technology. The combination of these single-instruction, multiple-data (SIMD) technologies will enable the development of advanced multimedia, signal processing, and modeling applications. To take advantage of the performance opportunities presented by these new capabilities, take into consideration the following:

- Ensure that your processor supports MMX technology, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2).
- Ensure that your operating system supports MMX technology and SSE (OS support for SSE2 is the same as OS support for SSE).
- Employ all of the optimization and scheduling strategies described in this book.
- Use stack and data alignment techniques to keep data properly aligned for efficient memory use.
- Utilize the cacheability instructions offered by SSE and SSE2.

This chapter gives an overview of the capabilities that allow you to better understand SIMD features and develop applications utilizing SIMD features of MMX technology, SSE, and SSE2.

Checking for Processor Support of SIMD Technologies

This section shows how to check whether a processor supports MMX technology, SSE, or SSE2. Once this check has been made, the appropriate SIMD technology can be included in your application in three ways:

1. Check for the SIMD technology during installation. If the desired SIMD technology is available, the appropriate DLLs can be installed.
2. Check for the SIMD technology during program execution and install the proper DLLs at runtime. This is effective for programs that may be executed on different machines.
3. Create a “fat” binary that includes multiple versions of routines; version that use the SIMD technology and versions that do not. Check for the SIMD technology during program execution and run the appropriate versions of the routines. This is also effective for programs that may be executed on different machines.

Checking for MMX Technology Support

Before you start coding with MMX technology, check if MMX technology is available on your system. Use the `cpuid` instruction to check the feature flags in the `edx` register. If `cpuid` returns bit 23 set to 1 in the feature flags, the processor supports MMX technology. Use the code segment in [Example 3-1](#) to load the feature flags in `edx` and test the result for the existence of MMX technology.

Example 3-1 Identification of MMX Technology with `cpuid`

```

...identify existence of cpuid instruction
...
...
...
mov eax, 1          ; request for feature flags
cpuid               ; 0Fh, 0A2h cpuid instruction
test edx, 00800000h ; is MMX technology bit (bit
                   ; 23) in feature flags equal to 1
jnz      Found

```


For more information on `cpuid` see, *Intel Processor Identification with CPUID Instruction*, order number 241618.

Checking for Streaming SIMD Extensions Support

Checking for support of Streaming SIMD Extensions (SSE) on your processor is similar to doing the same for MMX technology, but you must also check whether your operating system (OS) supports SSE. This is because the OS needs to manage saving and restoring the new state introduced by SSE for your application to properly function.

To check whether your system supports SSE, follow these steps:

1. Check that your processor supports the `cpuid` instruction and is a Pentium III or later processor.
2. Check the feature bits of `cpuid` for SSE existence.
3. Check for OS support for SSE.

[Example 3-2](#) shows how to find the SSE feature bit (bit 25) in the `cpuid` feature flags.

Example 3-2 Identification of SSE with `cpuid`

```
...identify existence of cpuid instruction
...                               ; identify processor
mov eax, 1                       ; request for feature flags
cpuid                            ; 0Fh, 0A2h    cpuid instruction
test EDX, 002000000h            ; bit 25 in feature flags equal to 1
jnz      Found
```

To find out whether the operating system supports SSE, simply execute a SSE instruction and trap for the exception if one occurs. An invalid opcode will be raised by the operating system and processor if either is not enabled for SSE. Catching the exception in a simple try/except clause (using structured exception handling in C++) and checking whether the exception code is an invalid opcode will give you the answer. See [Example 3-3](#).

Example 3-3 Identification of SSE by the OS

```
bool OSSupportCheck() {
    _try {
        __asm xorps xmm0, xmm0 ;Streaming SIMD Extension
    }
    _except(EXCEPTION_EXECUTE_HANDLER) {
        if (_exception_code()==STATUS_ILLEGAL_INSTRUCTION)
            /* SSE not supported */
            return (false);
    }
    /* SSE are supported by OS */
    return (true);
}
```

Checking for Streaming SIMD Extensions 2 Support

Checking for support of SSE2 on your processor is similar to that of SSE in that you must also check whether your operating system (OS) supports SSE. The OS requirements for SSE2 Support are the same as the requirements for SSE. To check whether your system supports SSE2, follow these steps:

1. Check that your processor has the `cpuid` instruction and is the Pentium 4 processor or later.
2. Check the feature bits of `cpuid` for SSE2 technology existence.
3. Check for OS support for SSE.

[Example 3-2](#) shows how to find the SSE2 feature bit (bit 25) in the `cpuid` feature flags.

Example 3-4 Identification of SSE2 with `cputid`

```

...identify existence of cputid instruction
...
mov eax, 1          ; identify processor
cputid              ; 0Fh, 0A2h  cputid instruction
test EDX, 00400000h ; bit 26 in feature flags equal to 1
jnz      Found

```

SSE2 require the same support from the operating system as SSE. To find out whether the operating system supports SSE2, simply execute a SSE2 instruction and trap for the exception if one occurs. An invalid opcode will be raised by the operating system and processor if either is not enabled for SSE2. Catching the exception in a simple try/except clause (using structured exception handling in C++) and checking whether the exception code is an invalid opcode will give you the answer. See [Example 3-3](#).

Example 3-5 Identification of SSE2 by the OS

```

bool OSSupportCheck() {
    _try {
        __asm xorpd xmm0, xmm0 ; SSE2}
    _except(EXCEPTION_EXECUTE_HANDLER) {
        if _exception_code()==STATUS_ILLEGAL_INSTRUCTION)
            /* SSE2not supported */
            return (false);
    }
    /* SSE2 are supported by OS */
    return (true);
}

```

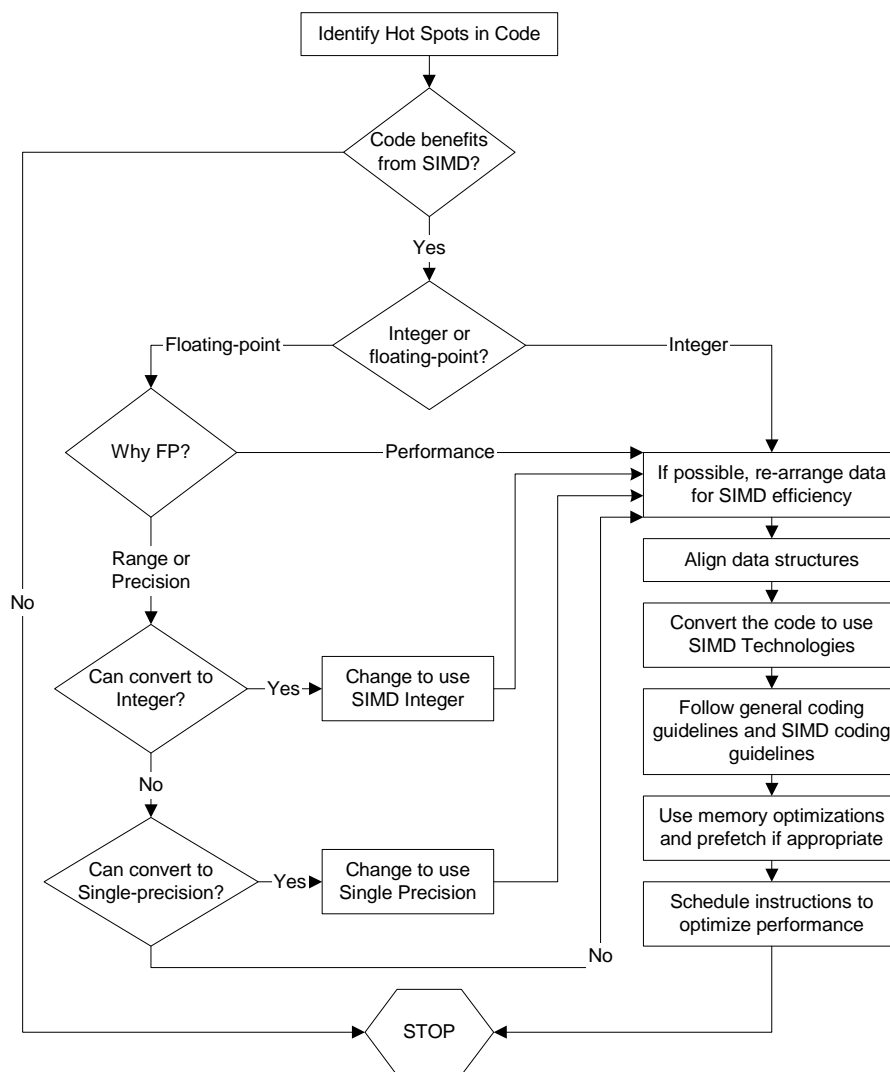
Considerations for Code Conversion to SIMD Programming

The VTune™ Performance Enhancement Environment CD provides tools to aid in the evaluation and tuning. But before you start implementing them, you need to know the answers to the following questions:

1. Will the current code benefit by using MMX technology, Streaming SIMD Extensions, or Streaming SIMD Extensions 2?
2. Is this code integer or floating-point?
3. What integer word size or floating-point precision do I need?
4. What coding techniques should I use?
5. What guidelines do I need to follow?
6. How should I arrange and align the datatypes?

[Figure 3-1](#) provides a flowchart for the process of converting code to MMX technology, Streaming SIMD Extensions, or Streaming SIMD Extensions 2.

Figure 3-1 Converting to Streaming SIMD Extensions Chart



To use any of the SIMD technologies optimally, you must evaluate the following situations in your code:

- fragments that are computationally intensive
- fragments that are executed often enough to have an impact on performance
- fragments that require integer computations with little data-dependent control flow
- fragments that require floating-point computations
- fragments that require help in using the cache hierarchy efficiently.

Identifying Hot Spots

To optimize performance, you can use the VTune™ Performance Analyzer to find the sections of code that occupy most of the computation time. Such sections are called the hotspots. For details on the VTune analyzer, see [“VTune™ Performance Analyzer”](#) in Appendix A. The VTune analyzer provides a hotspots view of a specific module to help you identify sections in your code that take the most CPU time and that have potential performance problems. For more explanation, see section [“Using Sampling Analysis for Optimization”](#) in Appendix A, which includes an example of a hotspots report. The hotspots view helps you identify sections in your code that take the most CPU time and that have potential performance problems.

The VTune analyzer enables you to change the view to show hotspots by memory location, functions, classes, or source files. You can double-click on a hotspot and open the source or assembly view for the hotspot and see more detailed information about the performance of each instruction in the hotspot.

The VTune analyzer offers focused analysis and performance data at all levels of your source code and can also provide advice at the assembly language level. The code coach analyzes and identifies opportunities for better performance of C/C++, Fortran and Java* programs, and suggests specific optimizations. Where appropriate, the coach displays pseudo-code to suggest the use of highly optimized intrinsics and functions in the Intel® Performance Library Suite. Because VTune analyzer is designed specifically for all of the Intel architecture (IA)-based processors, including the Pentium 4 processor, it can offer these detailed approaches to working with IA. See [“Code Coach Optimizations”](#) in Appendix A for more details and example of a code coach advice.

Determine If Code Benefits by Conversion to SIMD Execution

Identifying code that benefits by using SIMD technologies can be time-consuming and difficult. Likely candidates for conversion are applications that are highly computation intensive, such as the following:

- speech compression algorithms and filters
- speech recognition algorithms
- video display and capture routines
- rendering routines
- 3D graphics (geometry)
- image and video processing algorithms
- spatial (3D) audio
- physical modeling (graphics, CAD)
- workstation applications
- encryption algorithms

Generally, good candidate code is code that contains small-sized repetitive loops that operate on sequential arrays of integers of 8 or 16 bits for MMX technology, single-precision 32-bit floating-point data for SSE technology, or double precision 64-bit floating-point data for SSE2 (integer and floating-point data items should be sequential in memory). The repetitiveness of these loops incurs costly application processing time. However, these routines have potential for increased performance when you convert them to use one of the SIMD technologies.

Once you identify your opportunities for using a SIMD technology, you must evaluate what should be done to determine whether the current algorithm or a modified one will ensure the best performance.

Coding Techniques

The SIMD features of SSE2, SSE, and MMX technology require new methods of coding algorithms. One of them is vectorization. Vectorization is the process of transforming sequentially-executing, or scalar, code into code that can execute in parallel, taking advantage of the SIMD architecture parallelism. This section discusses the coding techniques available for an application to make use of the SIMD architecture.

To vectorize your code and thus take advantage of the SIMD architecture, do the following:

- Determine if the memory accesses have dependencies that would prevent parallel execution
- “Strip-mine” the loop to reduce the iteration count by the length of the SIMD operations (for example, four for single-precision floating-point SIMD, eight for 16-bit integer SIMD on the XMM registers)
- Re-code the loop with the SIMD instructions

Each of these actions is discussed in detail in the subsequent sections of this chapter. These sections also discuss enabling automatic vectorization via the Intel C++ Compiler.

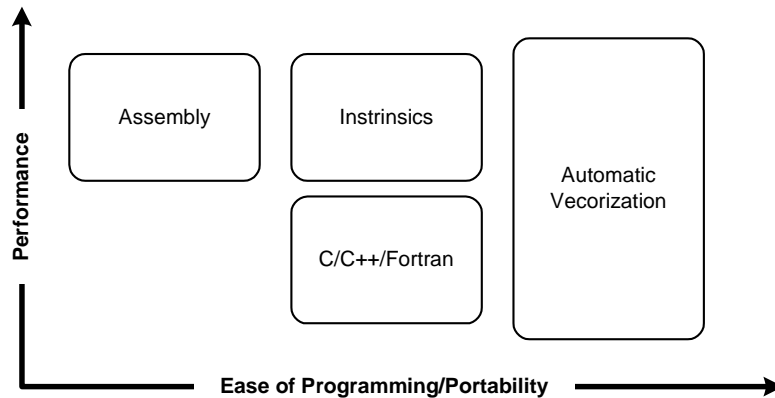
Coding Methodologies

Software developers need to compare the performance improvement that can be obtained from assembly code versus the cost of those improvements. Programming directly in assembly language for a target platform may produce the required performance gain, however, assembly code is not portable between processor architectures and is expensive to write and maintain.

Performance objectives can be met by taking advantage of the different SIMD technologies using high-level languages as well as assembly. The new C/C++ language extensions designed specifically for SSE2, SSE, and MMX technology help make this possible.

[Figure 3-2](#) illustrates the trade-offs involved in the performance of hand-coded assembly versus the ease of programming and portability.

Figure 3-2 Hand-Coded Assembly and High-Level Compiler Performance Trade-offs



The examples that follow illustrate the use of coding adjustments to enable the algorithm to benefit from the SSE. The same techniques may be used for single-precision floating-point, double-precision floating-point, and integer data under SSE2, SSE, and MMX technology.

As a basis for the usage model discussed in this section, consider a simple loop shown in [Example 3-6](#).

Example 3-6 Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Note that the loop runs for only four iterations. This allows a simple replacement of the code with Streaming SIMD Extensions.

For the optimal use of the Streaming SIMD Extensions that need data alignment on the 16-byte boundary, all examples in this chapter assume that the arrays passed to the routine, *a*, *b*, *c*, are aligned to 16-byte boundaries by a calling routine. For the methods to ensure this alignment, please refer to the application notes for the Pentium 4 processor available at <http://developer.intel.com>.

The sections that follow provide details on the coding methodologies: inlined assembly, intrinsics, C++ vector classes, and automatic vectorization.

Assembly

Key loops can be coded directly in assembly language using an assembler or by using inlined assembly (C-asm) in C/C++ code. The Intel compiler or assembler recognize the new instructions and registers, then directly generate the corresponding code. This model offers the opportunity for attaining greatest performance, but this performance is not portable across the different processor architectures.

[Example 3-7](#) shows the Streaming SIMD Extensions inlined assembly encoding.

Example 3-7 Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
    __asm {
        mov     eax, a
        mov     edx, b
        mov     ecx, c
        movaps  xmm0, XMMWORD PTR [eax]
        addps   xmm0, XMMWORD PTR [edx]
        movaps  XMMWORD PTR [ecx], xmm0
    }
}
```

Intrinsics

Intrinsics provide the access to the ISA functionality using C/C++ style coding instead of assembly language. Intel has defined three sets of intrinsic functions that are implemented in the Intel® C++ Compiler to support the MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. Four new C data types, representing 64-bit and 128-bit objects are used as the operands of these intrinsic functions. `__m64` is used for MMX integer SIMD, `__m128` is used for single-precision floating-point SIMD, `__m128i` is used for Streaming SIMD Extensions 2 integer SIMD and `__m128d` is used for double precision floating-point SIMD. These types enable the programmer to choose the implementation of an algorithm directly, while allowing the compiler to perform register allocation and instruction scheduling where possible. These intrinsics are portable among all Intel architecture-based processors supported by a compiler. The use of intrinsics allows you to obtain performance close to the levels achievable with assembly. The cost of writing and maintaining programs with intrinsics is considerably less. For a detailed description of the intrinsics and their use, refer to the *Intel C++ Compiler User's Guide*.

[Example 3-8](#) shows the loop from Example 3-4 using intrinsics.

Example 3-8 Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

The intrinsics map one-to-one with actual Streaming SIMD Extensions assembly code. The `xmmintrin.h` header file in which the prototypes for the intrinsics are defined is part of the Intel C++ Compiler included with the VTune™ Performance Enhancement Environment CD.

Intrinsics are also defined for the MMX technology ISA. These are based on the `__m64` data type to represent the contents of an `mm` register. You can specify values in bytes, short integers, 32-bit values, or as a 64-bit object.

The intrinsic data types, however, are not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

- Use intrinsic data types only on the left-hand side of an assignment as a return value or as a parameter. You cannot use it with other arithmetic expressions (for example, “+”, “>”).
- Use intrinsic data type objects in aggregates, such as unions to access the byte elements and structures; the address of an `__m64` object may be also used.
- Use intrinsic data type data only with the MMX technology intrinsics described in this guide.

For complete details of the hardware instructions, see the *Intel Architecture MMX Technology Programmer's Reference Manual*. For descriptions of data types, see the *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*.

Classes

A set of C++ classes has been defined and available in Intel C++ Compiler to provide both a higher-level abstraction and more flexibility for programming with MMX technology, Streaming SIMD Extensions and Streaming SIMD Extensions 2. These classes provide an easy-to-use and flexible interface to the intrinsic functions, allowing developers to write more natural C++ code without worrying about which intrinsic or assembly language instruction to use for a given operation. Since the intrinsic functions underlie the implementation of these C++ classes, the performance of applications using this methodology can approach that of one using the intrinsics. Further details on the use of these classes can be found in the *Intel C++ Class Libraries for SIMD Operations User's Guide*, order number 693500.

[Example 3-9](#) shows the C++ code using a vector class library. The example assumes the arrays passed to the routine are already aligned to 16-byte boundaries.

Example 3-9 C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
    F32vec4 *av=(F32vec4 *) a;
    F32vec4 *bv=(F32vec4 *) b;
    F32vec4 *cv=(F32vec4 *) c;

    *cv=*av + *bv;
}
```

Here, `fvec.h` is the class definition file and `F32vec4` is the class representing an array of four floats. The “+” and “=” operators are overloaded so that the actual Streaming SIMD Extensions implementation in the previous example is abstracted out, or hidden, from the developer. Note how much more this resembles the original code, allowing for simpler and faster programming.

Again, the example is assuming the arrays, passed to the routine, are already aligned to 16-byte boundary.

Automatic Vectorization

The Intel C++ Compiler provides an optimization mechanism by which simple loops, such as in [Example 3-6](#) can be automatically vectorized, or converted into Streaming SIMD Extensions code. The compiler uses similar techniques to those used by a programmer to identify whether a loop is suitable for conversion to SIMD. This involves determining whether the following might prevent vectorization:

- the layout of the loop and the data structures used
- dependencies amongst the data accesses in each iteration and across iterations

Once the compiler has made such a determination, it can generate vectorized code for the loop, allowing the application to use the SIMD instructions.

The caveat to this is that only certain types of loops can be automatically vectorized, and in most cases user interaction with the compiler is needed to fully enable this.

[Example 3-10](#) shows the code for automatic vectorization for the simple four-iteration loop (from [Example 3-6](#)).

Example 3-10 Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Compile this code using the `-Qax` and `-Qrestrict` switches of the Intel C++ Compiler, version 4.0 or later.

The `restrict` qualifier in the argument list is necessary to let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used, provides the only means of accessing the memory in question in the scope in which the pointers live. Without this qualifier, the compiler will not vectorize the loop because it cannot ascertain whether the array references in the loop overlap, and without this information, generating vectorized code is unsafe.

Refer to the *Intel® C++ Compiler User's Guide*, for more details on the use of automatic vectorization.

Stack and Data Alignment

To get the most performance out of code written for SIMD technologies data should be formatted in memory according to the guidelines described in this section. Assembly code with an unaligned accesses is a lot slower than an aligned access.

Alignment and Contiguity of Data Access Patterns

The new 64-bit packed data types defined by MMX technology, and the 128-bit packed data types for Streaming SIMD Extensions and Streaming SIMD Extensions 2 create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX technology and Streaming SIMD Extensions.

Using Padding to Align Data

However, when accessing SIMD data using SIMD operations, access to data can be improved simply by a change in the declaration. For example, consider a declaration of a structure, which represents a point in space plus an attribute.

```
typedef struct { short x,y,z; char a } Point;
Point pt[N];
```

Assume we will be performing a number of computations on *x*, *y*, *z* in three of the four elements of a SIMD word; see the [“Data Structure Layout”](#) section for an example. Even if the first element in array *pt* is aligned, the second element will start 7 bytes later and not be aligned (3 shorts at two bytes each plus a single byte = 7 bytes).

By adding the padding variable *pad*, the structure is now 8 bytes, and if the first element is aligned to 8 bytes (64 bits), all following elements will also be aligned. The sample declaration follows:

```
typedef struct { short x,y,z; char a; char pad; } Point;
Point pt[N];
```

Using Arrays to Make Data Contiguous

In the following code,

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

the second dimension *y* needs to be multiplied by a scaling value. Here the `for` loop accesses each *y* dimension in the array *pt* thus disallowing the access to contiguous data. This can degrade the performance of the application by increasing cache misses, by achieving poor utilization of each cache line that is fetched, and by increasing the chance for accesses which span multiple cache lines.

The following declaration allows you to vectorize the scaling operation and further improve the alignment of the data access patterns:

```
short ptx[N], pty[N], ptz[N];  
for (i=0; i<N; i++) pty[i] *= scale;
```

With the SIMD technology, choice of data organization becomes more important and should be made carefully based on the operations that will be performed on the data. In some applications, traditional data arrangements may not lead to the maximum performance.

A simple example of this is an FIR filter. An FIR filter is effectively a vector dot product in the length of the number of coefficient taps.

Consider the following code:

```
(data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of  
taps-1]*coeff [num of taps-1]),
```

If in the code above the filter operation of data element *i* is the vector dot product that begins at data element *j*, then the filter operation of data element *i+1* begins at data element *j+1*.

Assuming you have a 64-bit aligned data vector and a 64-bit aligned coefficients vector, the filter operation on the first data element will be fully aligned. For the second data element, however, access to the data vector will be misaligned. For an example of how to avoid the misalignment problem in the FIR filter, please refer to the application notes available at

<http://developer.intel.com/software/idap/processor/ia32/pentiumiii/sse.htm>.

Duplication and padding of data structures can be used to avoid the problem of data accesses in algorithms which are inherently misaligned. The [“Data Structure Layout”](#) section discusses further trade-offs for how data structures are organized.



CAUTION. *The duplication and padding technique overcomes the misalignment problem, thus avoiding the expensive penalty for misaligned data access, at the cost of increasing the data size. When developing your code, you should consider this tradeoff and use the option which gives the best performance.*

Stack Alignment For 128-bit SIMD Technologies

For best performance, the Streaming SIMD Extensions and Streaming SIMD Extensions 2 require their memory operands to be aligned to 16-byte (16B) boundaries. Unaligned data can cause significant performance penalties compared to aligned data. However, the existing software conventions for IA-32 (`stdcall`, `cdecl`, `fastcall`) as implemented in most compilers, do not provide any mechanism for ensuring that certain local data and certain parameters are 16-byte aligned. Therefore, Intel has defined a new set of IA-32 software conventions for alignment to support the new `__m128*` datatypes (`__m128`, `__m128d`, and `__m128i`) that meet the following conditions:

- Functions that use Streaming SIMD Extensions or Streaming SIMD Extensions 2 data need to provide a 16-byte aligned stack frame.
- The `__m128*` parameters need to be aligned to 16-byte boundaries, possibly creating “holes” (due to padding) in the argument block

These new conventions presented in this section as implemented by the Intel C++ Compiler can be used as a guideline for an assembly language code as well. In many cases, this section assumes the use of the `__m128*` data types, as defined by the Intel C++ Compiler, which represents an array of four 32-bit floats.

For more details on the stack alignment for Streaming SIMD Extensions and SSE2, see [Appendix D, “Stack Alignment”](#).

Data Alignment for MMX Technology

Many compilers enable alignment of variables using controls. This aligns the variables’ bit lengths to the appropriate boundaries. If some of the variables are not appropriately aligned as specified, you can align them using the C algorithm shown in [Example 3-11](#).

Example 3-11 C Algorithm for 64-bit Data Alignment

```
/* Make newp a pointer to a 64-bit aligned array */  
/* of NUM_ELEMENTS 64-bit elements. */  
double *p, *newp;  
p = (double*)malloc (sizeof(double)*(NUM_ELEMENTS+1));  
newp = (p+7) & (~0x7);
```

The algorithm in Example 3-11 aligns an array of 64-bit elements on a 64-bit boundary. The constant of 7 is derived from one less than the number of bytes in a 64-bit element, or 8-1. Aligning data in this manner avoids the significant performance penalties that can occur when an access crosses a cache line boundary.

Another way to improve data alignment is to copy the data into locations that are aligned on 64-bit boundaries. When the data is accessed frequently, this can provide a significant performance improvement.

Data Alignment for 128-bit data

Data must be 16-byte aligned when loading to or storing from the 128-bit XMM registers used by SSE and SSE2 to avoid severe performance penalties at best, and at worst, execution faults. Although there are move instructions (and intrinsics) to allow unaligned data to be copied into and out of the XMM registers when not using aligned data, such operations are much slower than aligned accesses. If, however, the data is not 16-byte-aligned and the programmer or the compiler does not detect this and uses the aligned instructions, a fault will occur. So, the rule is: keep the data 16-byte-aligned. Such alignment will also work for MMX technology code, even though MMX technology only requires 8-byte alignment. The following discussion and examples describe alignment techniques for Pentium 4 processor as implemented with the Intel C++ Compiler.

Compiler-Supported Alignment

The Intel C++ Compiler provides the following methods to ensure that the data is aligned.

Alignment by `F32vec4` or `__m128` Data Types. When compiler detects `F32vec4` or `__m128` data declarations or parameters, it will force alignment of the object to a 16-byte boundary for both global and local data, as well as parameters. If the declaration is within a function, the compiler will also align the function's stack frame to ensure that local data and parameters are 16-byte-aligned. For details on the stack frame layout that the compiler generates for both debug and optimized ("release"-mode) compilations, please refer to the relevant Intel application notes in the Intel Architecture Performance Training Center provided with the SDK.

The `__declspec(align(16))` specifications can be placed before data declarations to force 16-byte alignment. This is particularly useful for local or global data declarations that are assigned to 128-bit data types. The syntax for it is

```
__declspec(align(integer-constant))
```

where the *integer-constant* is an integral power of two but no greater than 32. For example, the following increases the alignment to 16-bytes:

```
__declspec(align(16)) float buffer[400];
```

The variable `buffer` could then be used as if it contained 100 objects of type `__m128` or `F32vec4`. In the code below, the construction of the `F32vec4` object, `x`, will occur with aligned data.

```
void foo() {  
    F32vec4 x = *(__m128 *) buffer;  
    ...  
}
```

Without the declaration of `__declspec(align(16))`, a fault may occur.

Alignment by Using a union Structure. Preferably, when feasible, a `union` can be used with 128-bit data types to allow the compiler to align the data structure by default. Doing so is preferred to forcing alignment with `__declspec(align(16))` because it exposes the true program intent to the compiler in that `__m128` data is being used. For example:

```
union {  
    float f[400];  
    __m128 m[100];  
} buffer;
```

The 16-byte alignment is used by default due to the `__m128` type in the `union`; it is not necessary to use `__declspec(align(16))` to force it.

In C++ (but not in C) it is also possible to force the alignment of a `class/struct/union` type, as in the code that follows:

```
struct __declspec(align(16)) my__m128  
{  
    float f[4];  
};
```

But, if the data in such a `class` is going to be used with the Streaming SIMD Extensions or Streaming SIMD Extensions 2, it is preferable to use a `union` to make this explicit. In C++, an anonymous `union` can be used to make this more convenient:

```
class my__m128 {  
    union {  
        __m128 m;  
        float f[4];  
    };  
};
```

In this example, because the `union` is anonymous, the names, `m` and `f`, can be used as immediate member names of `my__m128`. Note that `__declspec(align)` has no effect when applied to a `class`, `struct`, or `union` member in either C or C++.

Alignment by Using `__m64` or `double` Data. In some cases, for better performance, the compiler will align routines with `__m64` or `double` data to 16-bytes by default. The command-line switch, `-Qsfsalign16`, can be used to limit the compiler to only align in routines that contain 128-bit data. The default behavior is to use `-Qsfsalign8`, which instructs to align routines with 8- or 16-byte data types to 16-bytes.

For more details, see relevant Intel application notes in the Intel Architecture Performance Training Center provided with the SDK and the *Intel C++ Compiler User's Guide*.

Improving Memory Utilization

Memory performance can be improved by rearranging data and algorithms for SSE 2, SSE, and MMX technology intrinsics. The methods for improving memory performance involve working with the following:

- Data structure layout
- Strip-mining for vectorization and memory utilization
- Loop-blocking

Using the cacheability instructions, prefetch and streaming store, also greatly enhance memory utilization. For these instructions, see [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#).

Data Structure Layout

For certain algorithms, like 3D transformations and lighting, there are two basic ways of arranging the vertex data. The traditional method is the array of structures (AoS) arrangement, with a structure for each vertex (see Example 3-12). However this method does not take full advantage of the SIMD technology capabilities.

Example 3-12 AoS data structure

```
typedef struct{
    float x,y,z;
    int a,b,c;
    . . .
} Vertex;
Vertex Vertices[NumOfVertices];
```

The best processing method for code using SIMD technology is to arrange the data in an array for each coordinate (see Example 3-13). This data arrangement is called structure of arrays (SoA).

Example 3-13 SoA data structure

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int a[NumOfVertices];
    int b[NumOfVertices];
    int c[NumOfVertices];
    . . .
} VerticesList;
VerticesList Vertices;
```

There are two options for computing data in AoS format: perform operation on the data as it stands in AoS format, or re-arrange it (swizzle it) into SoA format dynamically. See [Example 3-14](#) for code samples of each option based on a dot-product computation.

Example 3-14 AoS and SoA Code Samples

```

; The dot product of an array of vectors (Array) and a
; fixed vector (Fixed) is a common operation in 3D
; lighting operations,
;   where Array = (x0,y0,z0),(x1,y1,z1),...
;   and Fixed = (xF,yF,zF)
; A dot product is defined as the scalar quantity
;           d0 = x0*xF + y0*yF + z0*zF.

; AoS code
; All values marked DC are "don't-care."
; In the AOS model, the vertices are stored in the
; xyz format
movaps  xmm0, Array      ; xmm0 = DC, x0,   y0,   z0
movaps  xmm1, Fixed      ; xmm1 = DC, xF,   yF,   zF
mulps   xmm0, xmm1      ; xmm0 = DC, x0*xF, y0*yF, z0*zF
movhlps xmm1, xmm0      ; xmm1 = DC, DC,   DC,   x0*xF
addps   xmm1, xmm0      ; xmm0 = DC, DC,   DC,
;                          x0*xF+z0*zF

movaps  xmm2, xmm1
shufps  xmm2, xmm2, 55h  ; xmm2 = DC, DC,   DC,   y0*yF
addps   mm2, xmm1      ; xmm1 = DC, DC,   DC,
;                          x0*xF+y0*yF+z0*zF

; SoA code
;
; X = x0,x1,x2,x3
; Y = y0,y1,y2,y3
; Z = z0,z1,z2,z3

```

continued

Example 3-14 AoS and SoA Code Samples (continued)

```

; A = xF, xF, xF, xF
; B = yF, yF, yF, yF
; C = zF, zF, zF, zF

movaps xmm0, X      ; xmm0 = x0, x1, x2, x3
movaps xmm1, Y      ; xmm0 = y0, y1, y2, y3
movaps xmm2, Z      ; xmm0 = z0, z1, z2, z3
mulps  xmm0, A      ; xmm0 = x0*xF, x1*xF, x2*xF, x3*xF
mulps  xmm1, B      ; xmm1 = y0*yF, y1*yF, y2*yF, y3*yF
mulps  xmm2, C      ; xmm2 = z0*zF, z1*zF, z2*zF, z3*zF
addps  xmm0, xmm1
addps  xmm0, xmm2    ; xmm0 = (x0*xF+y0*yF+z0*zF), ...

```

Performing SIMD operations on the original AoS format can require more calculations and some of the operations do not take advantage of all of the SIMD elements available. Therefore, this option is generally less efficient.

The recommended way for computing data in AoS format is to swizzle each set of elements to SoA format before processing it using SIMD technologies. This swizzling can either be done dynamically during program execution or statically when the data structures are generated; see Chapters 4 and 5 for specific examples of swizzling code. Performing the swizzle dynamically is usually better than using AoS, but is somewhat inefficient as there is the overhead of extra instructions during computation. Performing the swizzle statically, when the data structures are being laid out, is best as there is no runtime overhead.

As mentioned earlier, the SoA arrangement allows more efficient use of the parallelism of the SIMD technologies because the data is ready for computation in a more optimal vertical manner: multiplying components x_0, x_1, x_2, x_3 by xF, xF, xF, xF using 4 SIMD execution slots to produce 4 unique results. In contrast, computing directly on AoS data can lead to horizontal operations that consume SIMD execution slots but produce only a single scalar result as shown by the many “don’t-care” (DC) slots in [Example 3-14](#).

Use of the SoA format for data structures can also lead to more efficient use of caches and bandwidth. When the elements of the structure are not accessed with equal frequency, such as when element x , y , z are accessed ten times more often than the other entries, then SoA not only saves memory, but it also prevents fetching unnecessary data items a , b , and c .

Example 3-15 Hybrid SoA data structure

```

NumOfGroups = NumOfVertices/SIMDwidth
typedef struct{
    float x[SIMDwidth];
    float y[SIMDwidth];
    float z[SIMDwidth];
} VerticesCoordList;
typedef struct{
    int a[SIMDwidth];
    int b[SIMDwidth];
    int c[SIMDwidth];
    . . .
} VerticesColorList;
VerticesCoordList VerticesCoord[NumOfGroups];
VerticesColorList VerticesColor[NumOfGroups];

```

Note that SoA can have the disadvantage of requiring more independent memory stream references. A computation that uses arrays x , y , and z in [Example 3-13](#) would require three separate data streams. This can require the use of more prefetches, additional address generation calculations, as well as having a greater impact on DRAM page access efficiency. An alternative, a hybrid SoA approach blends the two alternatives (see [Example 3-15](#)). In this case, only 2 separate address streams are generated and referenced: one which contains $xxxx, yyyy, zzzz, zzzz, \dots$ and the other which contains $aaaa, bbbb, cccc, aaaa, dddd, \dots$. This also prevents fetching

unnecessary data, assuming the variables x , y , z are always used together; whereas the variables a , b , c would also be used together, but not at the same time as x , y , z . This hybrid SoA approach ensures:

- data is organized to enable more efficient vertical SIMD computation,
- simpler/less address generation than AoS,
- fewer streams, which reduces DRAM page misses,
- use of fewer prefetches, due to fewer streams,
- efficient cache line packing of data elements that are used concurrently.

With the advent of the SIMD technologies, the choice of data organization becomes more important and should be carefully based on the operations to be performed on the data. This will become increasingly important in the Pentium 4 processor and future processors. In some applications, traditional data arrangements may not lead to the maximum performance. Application developers are encouraged to explore different data arrangements and data segmentation policies for efficient computation. This may mean using a combination of AoS, SoA, and Hybrid SoA in a given application.

Strip Mining

Strip mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure twofold:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each “vector,” or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed. Consider [Example 3-16](#).

Example 3-16 Pseudo-code Before Strip Mining

```
typedef struct _VERTEX {
    float x, y, z, nx, ny, nz, u, v;
} Vertex_rec;

main()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i<Num; i++) {
        Transform(v[i]);
    }
    for (i=0; i<Num; i++) {
        Lighting(v[i]);
    }
    ....
}
```

The main loop consists of two functions: transformation and lighting. For each object, the main loop calls a transformation routine to update some data, then calls the lighting routine to further work on the data. If the size of array `v[Num]` is larger than the cache, then the coordinates for `v[i]` that were cached during `Transform(v[i])` will be evicted from the cache by the time we do `Lighting(v[i])`. This means that `v[i]` will have to be fetched from main memory a second time, reducing performance.

Example 3-17 Strip Mined Code

```
main()
{
    Vertex_rec v[Num];
    ....
    for (i=0; i < Num; i+=strip_size) {
        for (j=i; j < min(Num, i+strip_size); j++) {
            Transform(v[j]);
        }
        for (j=i; j < min(Num, i+strip_size); j++) {
            Lighting(v[j]);
        }
    }
}
```

In [Example 3-17](#), the computation has been strip-mined to a size `strip_size`. The value `strip_size` is chosen such that `strip_size` elements of array `v[Num]` fit into the cache hierarchy. By doing this, a given element `v[i]` brought into the cache by `Transform(v[i])` will still be in the cache when we perform `Lighting(v[i])`, and thus improve performance over the non-strip-mined code.

Loop Blocking

Loop blocking is another useful technique for memory performance optimization. The main purpose of loop blocking is also to eliminate as many cache misses as possible. This technique transforms the memory domain of a given problem into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse. In fact, one can treat loop blocking as strip mining in two or more dimensions. Consider the code in [Example 3-16](#) and access pattern in [Figure 3-3](#). The two-dimensional array `A` is referenced in the `j` (column) direction and then referenced in the `i` (row) direction (column-major order); whereas array `B` is

referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array A and B for the code in [Example 3-18](#) would be 1 and MAX, respectively.

Example 3-18 Loop Blocking

A. Original loop

```
float A[MAX, MAX], B[MAX, MAX]
for (i=0; i< MAX; i++) {
    for (j=0; j< MAX; j++) {
        A[i,j] = A[i,j] + B[j, i];
    }
}
```

B. Transformed Loop after Blocking

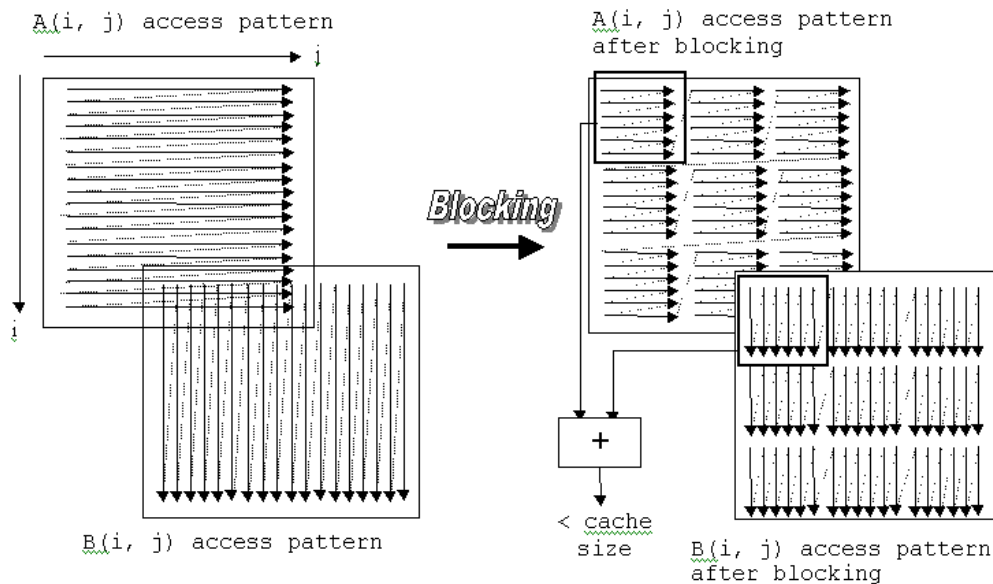
```
float A[MAX, MAX], B[MAX, MAX];
for (i=0; i< MAX; i+=block_size) {
    for (j=0; j< MAX; j+=block_size) {
        for (ii=i; ii<i+block_size; ii++) {
            for (jj=j; jj<j+block_size; jj++) {
                A[ii,jj] = A[ii,jj] + B[jj, ii];
            }
        }
    }
}
```

For the first iteration of the inner loop, each access to array B will generate a cache miss. If the size of one row of array A, that is, A[2, 0:MAX-1], is large enough, by the time the second iteration starts, each access to array B will always generate a cache miss. For instance, on the first iteration, the cache line containing B[0, 0:7] will be brought in when B[0,0] is referenced because the float type variable is four bytes and each cache line is 32 bytes. Due to the limitation of cache capacity, this line will be evicted due to conflict misses before the inner loop reaches the end. For the next iteration of

the outer loop, another cache miss will be generated while referencing $B[0, 1]$. In this manner, a cache miss occurs when each element of array B is referenced, that is, there is no data reuse in the cache at all for array B .

This situation can be avoided if the loop is blocked with respect to the cache size. In [Figure 3-3](#), a `block_size` is selected as the loop blocking factor. Suppose that `block_size` is 8, then the blocked chunk of each array will be eight cache lines (32 bytes each). In the first iteration of the inner loop, $A[0, 0:7]$ and $B[0, 0:7]$ will be brought into the cache. $B[0, 0:7]$ will be completely consumed by the first iteration of the outer loop. Consequently, $B[0, 0:7]$ will only experience one cache miss after applying loop blocking optimization in lieu of eight misses for the original algorithm. As illustrated in [Figure 3-3](#), arrays A and B are blocked into smaller rectangular chunks so that the total size of two blocked A and B chunks is smaller than the cache size. This allows maximum data reuse.

Figure 3-3 Loop Blocking Access Pattern



As one can see, all the redundant cache misses can be eliminated by applying this loop blocking technique. If MAX is huge, loop blocking can also help reduce the penalty from DTLB (data translation look-aside buffer) misses. In addition to improving the cache/memory performance, this optimization technique also saves external bus bandwidth.

Instruction Selection

The following section gives some guidelines for choosing instructions to complete a task.

One barrier to SIMD computation can be the existence of data-dependent branches. Conditional moves can be used to eliminate data-dependent branches. Conditional moves can be emulated in SIMD computation by using masked compares and logicals, as shown in [Example 3-19](#).

Example 3-19 Emulation of Conditional Moves

High-level code:

```
short A[MAX_ELEMENT], B[MAX_ELEMENT], C[MAX_ELEMENT], D[MAX_ELEMENT],  
E[MAX_ELEMENT];
```

```
for (i=0; i<MAX_ELEMENT; i++) {  
    if (A[i] > B[i]) {  
        C[i] = D[i];  
    } else {  
        C[i] = E[i];  
    }  
}
```

Assembly code:

```
xor     eax, eax
```

continued

Example 3-19 Emulation of Conditional Moves (continued)

```
top_of_loop:
    movq    mm0, [A + eax]
    pcmptgtw mm0, [B + eax]; Create compare mask
    movq    mm1, [D + eax]
    pand    mm1, mm0; Drop elements where A<B
    pandn   mm0, [E + eax] ; Drop elements where A>B
    por     mm0, mm1; Crete single word
    movq    [C + eax], mm0

    add     eax, 8
    cmp     eax, MAX_ELEMENT*2
    jle     top_of_loop
```

Note that this can be applied to both SIMD integer and SIMD floating-point code.

If there are multiple consumers of an instance of a register, group the consumers together as closely as possible. However, the consumers should not be scheduled near the producer.

Tuning the Final Application

The best way to tune your application once it is functioning correctly is to use a profiler that measures the application while it is running on a system. VTune analyzer can help you determine where to make changes in your application to improve performance. Using the VTune analyzer can help you with various phases required for optimized performance. See “[VTune™ Performance Analyzer](#)” in Appendix A for more details on how to use the VTune analyzer. After every effort to optimize, you should check the performance gains to see where you are making your major optimization gains.

Optimizing for SIMD Integer Applications

4

The SIMD integer instructions provide performance improvements in applications that are integer-intensive and can take advantage of the SIMD architecture of Intel Pentium II, Pentium III, and Pentium 4 processors.

The guidelines for using these instructions in addition to the guidelines described in [Chapter 2](#), will help develop fast and efficient code that scales well across all processors with MMX™ technology, processors that use Streaming SIMD Extensions (SSE) SIMD integer instructions, as well as the Pentium 4 processor with the SIMD integer instructions in the Streaming SIMD Extensions 2 (SSE2).

For the sake of brevity, the collection of 64-bit and 128-bit SIMD integer instructions supported by MMX technology, SSE, and SSE2 shall be referred to as SIMD integer instructions.

Unless otherwise noted, the following sequences are written for the 64-bit integer registers. Note that they can easily be changed to use the 128-bit SIMD integer form available with SSE2 by replacing the references to `mm0-mm7` with references to `xmm0-xmm7`.

This chapter contains several simple examples that will help you to get started with coding your application. The goal is to provide simple, low-level operations that are frequently used. The examples use a minimum number of instructions necessary to achieve best performance on the Pentium, Pentium Pro, Pentium II, Pentium III and Pentium 4 processors.

Each example includes a short description, sample code, and notes if necessary. These examples do not address scheduling as it is assumed the examples will be incorporated in longer code sequences.

For planning considerations of using the new SIMD integer instructions, refer to [“Checking for Streaming SIMD Extensions 2 Support”](#) in Chapter 3.

General Rules on SIMD Integer Code

The overall rules and suggestions are as follows:

- Do not intermix 64-bit SIMD integer instructions with x87 floating-point instructions. See [“Using SIMD Integer with x87 Floating-point”](#) section. Note that all of the SIMD integer instructions can be intermixed without penalty.
- When writing SSE2 code that works with both integer and floating-point data, use the subset of SIMD convert instructions or load/store instructions to ensure that the input operands in XMM registers contain properly defined data type to match the instruction. Code sequences containing cross-typed usage will produce the same result across different implementations, but will incur a significant performance penalty. Using SSE or SSE2 instructions to operate on type-mismatched SIMD data in the XMM register is strongly discouraged.
- Use the optimization rules and guidelines described in Chapters 2 and 3 that apply both to the Pentium 4 processor in general and to using the SIMD integer instructions.
- Incorporate the prefetch instruction whenever possible (for details, refer to [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#)).
- Emulate conditional moves by using masked compares and logicals instead of using conditional branches.

Using SIMD Integer with x87 Floating-point

All 64-bit SIMD integer instructions use the MMX registers, which share register state with the x87 floating-point stack. Because of this sharing, certain rules and considerations apply. Instructions which use the MMX registers cannot be freely intermixed with x87 floating-point registers. Care must be taken when switching between using 64-bit SIMD integer instructions and x87 floating-point instructions (see “Using the EMMS Instruction” section below).

The SIMD floating-point operations and 128-bit SIMD integer operations can be freely intermixed with either x87 floating-point operations or 64-bit SIMD integer operations. The SIMD floating-point operations and 128-bit SIMD integer operations

use registers that are unrelated to the x87 FP / MMX registers. The `emms` instruction is not needed to transition to or from SIMD floating-point operations or 128-bit SIMD operations.

Using the EMMS Instruction

When generating 64-bit SIMD integer code, keep in mind that the eight MMX registers are aliased on the x87 floating-point registers. Switching from MMX instructions to x87 floating-point instructions incurs a finite delay, so it is the best to minimize switching between these instruction types. But when you need to switch, the `emms` instruction provides an efficient means to clear the x87 stack so that subsequent x87 code can operate properly on the x87 stack.

As soon as any instruction makes reference to an MMX register, all valid bits in the x87 floating-point tag word are set, which implies that all x87 registers contain valid values. In order for software to operate correctly, the x87 floating-point stack should be emptied when starting a series of x87 floating-point calculations after operating on the MMX registers

Using `emms` clears all of the valid bits, effectively emptying the x87 floating-point stack and making it ready for new x87 floating-point operations. The `emms` instruction ensures a clean transition between using operations on the MMX registers and using operations on the x87 floating-point stack. On the Pentium 4 processor, there is a finite overhead for using the `emms` instruction.

Failure to use the `emms` instruction (or the `_mm_empty()` intrinsic) between operations on the MMX registers and operations on the x87 floating-point registers may lead to unexpected results.



CAUTION. *Failure to reset the tag word for FP instructions after using an MMX instruction can result in faulty execution or poor performance.*

Guidelines for Using EMMS Instruction

When developing code with both x87 floating-point and 64-bit SIMD integer instructions, follow these steps:

1. Always call the `emms` instruction at the end of 64-bit SIMD integer code when the code transitions to x87 floating-point code.
2. Insert the `emms` instruction at the end of all 64-bit SIMD integer code segments to avoid an x87 floating-point stack overflow exception when an x87 floating-point instruction is executed.

When writing an application that uses both floating-point and 64-bit SIMD integer instructions, use the following guidelines to help you determine when to use `emms`:

- *If next instruction is x87 FP:* Use `_mm_empty()` after a 64-bit SIMD integer instruction if the next instruction is an x87 FP instruction; for example, before doing calculations on floats, doubles or long doubles.
- *Don't empty when already empty:* If the next instruction uses an MMX register, `_mm_empty()` incurs a cost with no benefit.
- *Group Instructions:* Try to partition regions that use x87 FP instructions from those that use 64-bit SIMD integer instructions. This eliminates needing an `emms` instruction within the body of a critical loop.
- *Runtime initialization:* Use `_mm_empty()` during runtime initialization of `__m64` and x87 FP data types. This ensures resetting the register between data type transitions. See [Example 4-1](#) for coding usage.

Example 4-1 Resetting the Register between `__m64` and FP Data Types

Incorrect Usage

```
__m64 x = _m_padd(y, z);  
float f = init();
```

Correct Usage

```
__m64 x = _m_padd(y, z);  
float f = (_mm_empty(), init());
```

Further, you must be aware that your code generates an MMX instruction, which uses the MMX registers with the Intel C++ Compiler, in the following situations:

- when using a 64-bit SIMD integer intrinsic from MMX technology, SSE, or SSE2
- when using a 64-bit SIMD integer instruction from MMX technology, SSE, or SSE2 through inline assembly
- when referencing an `__m64` data type variable

Additional information on the x87 floating-point programming model can be found in the *IA-32 Intel® Architecture Software Developer's Manual*, Volume 1. For more documentation on `emms`, visit the <http://developer.intel.com> web site.

Data Alignment

Make sure that 64-bit SIMD integer data is 8-byte aligned and that 128-bit SIMD integer data is 16-byte aligned. Referencing unaligned 64-bit SIMD integer data can incur a performance penalty due to accesses that span 2 cache lines. Referencing unaligned 128-bit SIMD integer data will result in an exception unless the `movdqu` (move double-quadword unaligned) instruction is used. Using the `movdqu` instruction on unaligned data can result in lower performance than using 16-byte aligned references.

Refer to section “[Stack and Data Alignment](#)” in Chapter 3 for more information.

Data Movement Coding Techniques

In general, better performance can be achieved if the data is pre-arranged for SIMD computation (see the “[Improving Memory Utilization](#)” section of Chapter 3). However, this may not always be possible. This section covers techniques for gathering and re-arranging data for more efficient SIMD computation.

Unsigned Unpack

The MMX technology provides several instructions that are used to pack and unpack data in the MMX registers. The unpack instructions can be used to zero-extend an unsigned number. [Example 4-2](#) assumes the source is a packed-word (16-bit) data type.

Example 4-2 Unsigned Unpack Instructions

```

; Input:
;          MM0          source value
;          MM7 0        a local variable can be used
;                       instead of the register MM7 if
;                       desired.
; Output:
;          MM0          two zero-extended 32-bit
;                       doublewords from two low-end
;                       words
;          MM1          two zero-extended 32-bit
;                       doublewords from two high-end
;                       words

movq      MM1, MM0      ; copy source
punpcklwd MM0, MM7      ; unpack the 2 low-end words
; into two 32-bit doubleword
punpckhwd MM1, MM7      ; unpack the 2 high-end words
; into two 32-bit doublewords

```

Signed Unpack

Signed numbers should be sign-extended when unpacking the values. This is similar to the zero-extend shown above except that the `psrad` instruction (packed shift right arithmetic) is used to effectively sign extend the values. [Example 4-3](#) assumes the source is a packed-word (16-bit) data type.

Example 4-3 Signed Unpack Code

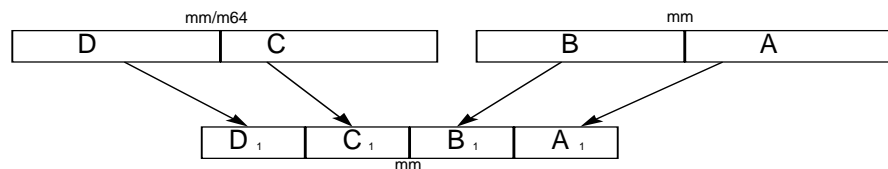
```

; Input:
;          MM0          source value
; Output:
;          MM0          two sign-extended 32-bit doublewords
;                      from the two low-end words
;          MM1          two sign-extended 32-bit doublewords
;                      from the two high-end words
;
movq      MM1, MM0      ; copy source
punpcklwd MM0, MM0      ; unpack the 2 low end words of the source
;                      ; into the second and fourth words of the
;                      ; destination
punpckhwd MM1, MM1      ; unpack the 2 high-end words of the source
;                      ; into the second and fourth words of the
;                      ; destination
psrad     MM0, 16       ; sign-extend the 2 low-end words of the source
;                      ; into two 32-bit signed doublewords
psrad     MM1, 16       ; sign-extend the 2 high-end words of the
;                      ; source into two 32-bit signed doublewords

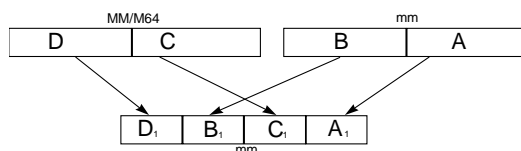
```

Interleaved Pack with Saturation

The pack instructions pack two values into the destination register in a predetermined order. Specifically, the `packssdw` instruction packs two signed doublewords from the source operand and two signed doublewords from the destination operand into four signed words in the destination register as shown in [Figure 4-1](#).

Figure 4-1 `PACKSSDW mm, mm/mm64` Instruction Example

[Figure 4-2](#) illustrates two values interleaved in the destination register, and [Example 4-4](#) shows code that uses the operation. The two signed doublewords are used as source operands and the result is interleaved signed words. The pack instructions can be performed with or without saturation as needed.

Figure 4-2 Interleaved Pack with Saturation

Example 4-4 Interleaved Pack with Saturation

```

; Input:
;           MM0          signed source1 value
;           MM1          signed source2 value
; Output:
;           MM0          the first and third words contain the
;                           signed-saturated doublewords from MM0,
;                           the second and fourth words contain
;                           signed-saturated doublewords from MM1
;
packssdw    MM0, MM0      ; pack and sign saturate
packssdw    MM1, MM1      ; pack and sign saturate
punpcklwd   MM0, MM1      ; interleave the low-end 16-bit
                           ; values of the operands

```

The pack instructions always assume that the source operands are signed numbers. The result in the destination register is always defined by the pack instruction that performs the operation. For example, the `packssdw` instruction packs each of the two signed 32-bit values of the two sources into four saturated 16-bit signed values in the destination register. The `packuswb` instruction, on the other hand, packs each of the four signed 16-bit values of the two sources into eight saturated eight-bit unsigned values in the destination. A complete specification of the MMX instruction set can be found in the *Intel® Architecture MMX™ Technology Programmer's Reference Manual*, order number 243007.

Interleaved Pack without Saturation

[Example 4-5](#) is similar to Example 4-4 except that the resulting words are not saturated. In addition, in order to protect against overflow, only the low order 16 bits of each doubleword are used in this operation.

Example 4-5 Interleaved Pack without Saturation

```

; Input:
;      MM0      signed source value
;      MM1      signed source value
; Output:
;      MM0      the first and third words contain the
;               low 16-bits of the doublewords in MM0,
;               the second and fourth words contain the
;               low 16-bits of the doublewords in MM1

pslld  MM1, 16    ; shift the 16 LSB from each of the
                  ; doubleword values to the 16 MSB
                  ; position

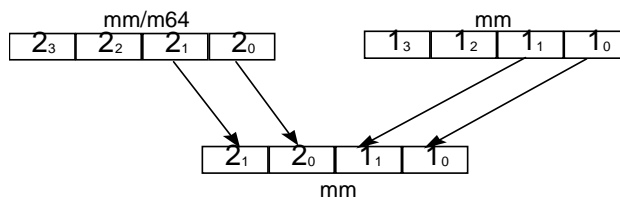
pand   MM0, {0,fff,0,fff}
                  ; mask to zero the 16 MSB
                  ; of each doubleword value

por    MM0, MM1   ; merge the two operands

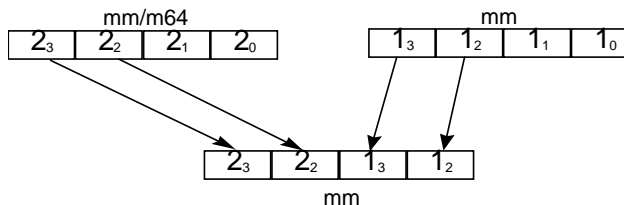
```

Non-Interleaved Unpack

The unpack instructions perform an interleave merge of the data elements of the destination and source operands into the destination register. The following example merges the two operands into the destination registers without interleaving. For example, take two adjacent elements of a packed-word data type in `source1` and place this value in the low 32 bits of the results. Then take two adjacent elements of a packed-word data type in `source2` and place this value in the high 32 bits of the results. One of the destination registers will have the combination illustrated in [Figure 4-3](#).

Figure 4-3 Result of Non-Interleaved Unpack Low in MM0

The other destination register will contain the opposite combination illustrated in [Figure 4-4](#).

Figure 4-4 Result of Non-Interleaved Unpack High in MM1

Code in the [Example 4-6](#) unpacks two packed-word sources in a non-interleaved way. The goal is to use the instruction which unpacks doublewords to a quadword, instead of using the instruction which unpacks words to doublewords.

Example 4-6 Unpacking Two Packed-word Sources in a Non-interleaved Way

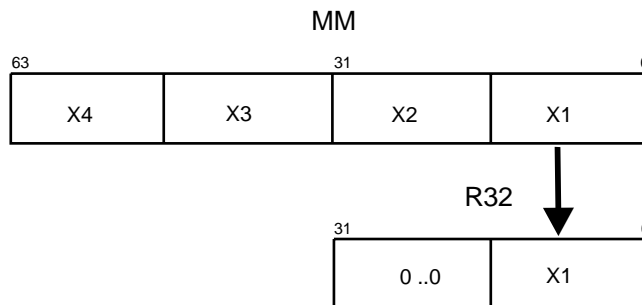
```

; Input:
;          MM0          packed-word source value
;          MM1          packed-word source value
; Output:
;          MM0          contains the two low-end words of the
;                      original sources, non-interleaved
;          MM2          contains the two high end words of the
;                      original sources, non-interleaved.
movq      MM2, MM0      ; copy source1
punpckldq MM0, MM1      ; replace the two high-end words
                        ; of MM0 with two low-end words of
                        ; MM1; leave the two low-end words
                        ; of MM0 in place
punpckhdq MM2, MM1      ; move two high-end words of MM2
                        ; to the two low-end words of MM2;
                        ; place the two high-end words of
                        ; MM1 in two high-end words of MM2

```

Extract Word

The `pextrw` instruction takes the word in the designated MMX register selected by the two least significant bits of the immediate value and moves it to the lower half of a 32-bit integer register, see [Figure 4-5](#) and [Example 4-7](#).

Figure 4-5 `pextrw` Instruction**Example 4-7** `pextrw` Instruction Code

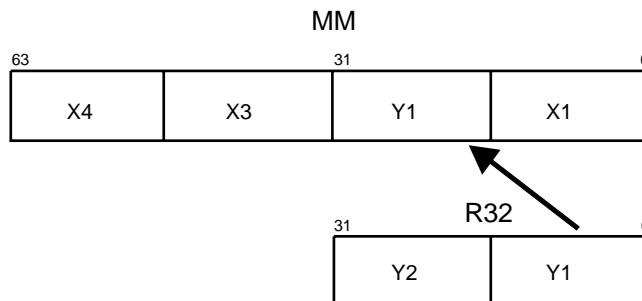
```

; Input:
;     eax     source value
;             immediate value:"0"
; Output:
;     edx     32-bit integer register containing the
;             extracted word in the low-order bits &
;             the high-order bits zero-extended
movq    mm0, [eax]
pextrw  edx, mm0, 0

```

Insert Word

The `pinsrw` instruction loads a word from the lower half of a 32-bit integer register or from memory and inserts it in the MMX technology destination register at a position defined by the two least significant bits of the immediate constant. Insertion is done in such a way that the three other words from the destination register are left untouched, see [Figure 4-6](#) and [Example 4-8](#).

Figure 4-6 `pinsrw` Instruction**Example 4-8** `pinsrw` Instruction Code

```

; Input:
;     edx    pointer to source value
; Output:
;     mm0     register with new 16-bit value inserted
;
mov     eax, [edx]
pinsrw  mm0, eax, 1

```

If all of the operands in a register are being replaced by a series of `pinsrw` instructions, it can be useful to clear the content and break the dependence chain by either using the `pxor` instruction or loading the register. See the [“Clearing Registers”](#) section in Chapter 2.

Example 4-9 Repeated pinsrw Instruction Code

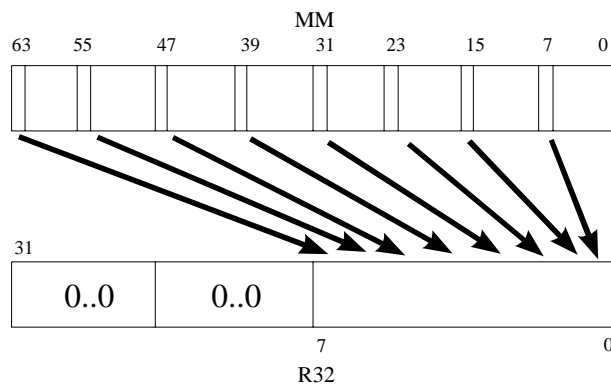
```

; Input:
;         edx         pointer to structure containing source
;                     values at offsets: of +0, +10, +13, and +24
;                     immediate value: "1"
; Output:
;         MMX         register with new 16-bit value inserted
;
pxor      mm0, mm0    ; Breaks dependedncy on previous value of mm0
mov       eax, [edx]
pinsrw    mm0, eax, 0
mov       eax, [edx+10]
pinsrw    mm0, eax, 1
mov       eax, [edx+13]
pinsrw    mm0, eax, 2
mov       eax, [edx+24]
pinsrw    mm0, eax, 3

```

Move Byte Mask to Integer

The `pmovmskb` instruction returns a bit mask formed from the most significant bits of each byte of its source operand. When used with the 64-bit MMX registers, this produces an 8-bit mask, zeroing out the upper 24 bits in the destination register. When used with the 128-bit XMM registers, it produces a 16-bit mask, zeroing out the upper 16 bits in the destination register. The 64-bit version is shown in [Figure 4-7](#) and [Example 4-10](#).

Figure 4-7 `pmovmskb` Instruction Example**Example 4-10** `pmovmskb` Instruction Code

```

; Input:
;     source value
; Output:
;     32-bit register containing the byte mask in the lower
;     eight bits
;
movq    mm0, [edi]
pmovmskb eax, mm0

```

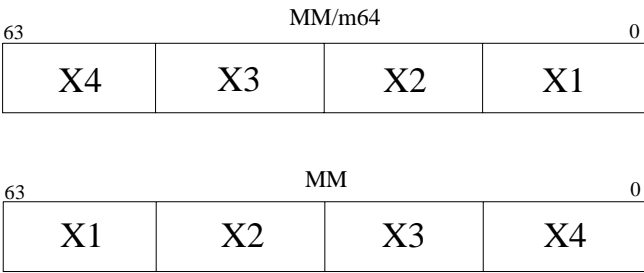

Packed Shuffle Word for 64-bit Registers

The `pshuf` instruction (see [Figure 4-8](#), [Example 4-11](#)) uses the immediate (`imm8`) operand to select between the four words in either two MMX registers or one MMX register and a 64-bit memory location. Bits 1 and 0 of the immediate value encode the source for destination word 0 in MMX register (`[15:0]`), and so on as shown in the table:

Bits	Word
1 - 0	0
3 - 2	1
5 - 4	2
7 - 6	3

Bits 7 and 6 encode for word 3 in MMX register (`[63:48]`). Similarly, the 2-bit encoding represents which source word is used, for example, binary encoding of 10 indicates that source word 2 in MMX register/memory (`mm/mem[47:32]`) is used, see [Figure 4-8](#) and [Example 4-11](#).

Figure 4-8 `pshuf` Instruction Example



Example 4-11 pshuf Instruction Code

```

; Input:
;      edi      source value
; Output:
;      MM1      MM register containing re-arranged words
movq   mm0, [edi]
pshufw mm1, mm0, 0x1b

```

Packed Shuffle Word for 128-bit Registers

The `pshufw/pshufhw` instruction performs a full shuffle of any source word field within the low/high 64 bits to any result word field in the low/high 64 bits, using an 8-bit immediate operand; the other high/low 64 bits are passed through from the source operand.

The `pshufd` instruction performs a full shuffle of any double-word field within the 128-bit source to any double-word field in the 128-bit result, using an 8-bit immediate operand.

No more than 3 instructions, using `pshufw/pshufhw/pshufd`, are required to implement some common data shuffling operations. Broadcast, Swap, and Reverse are illustrated in [Example 4-12](#), [Example 4-13](#), and [Example 4-14](#), respectively.

Example 4-12 Broadcast using 2 instructions

```

/* Goal:  Broadcast the value from word 5 to all words */
/* Instruction Result */
      | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFHW (3,2,1,1) | 7| 6| 5| 5| 3| 2| 1| 0|

PSHUFD (2,2,2,2) | 5| 5| 5| 5| 5| 5| 5| 5|

```

Example 4-13 Swap using 3 instructions

```

/* Goal:  Swap the values in word 6 and word 1 */
/* Instruction Result */
      | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFD (3,0,1,2)| 7| 6| 1| 0| 3| 2| 5| 4|

PSHUFHW (3,1,2,0)| 7| 1| 6| 0| 3| 2| 5| 4|

PSHUFD (3,0,1,2)| 7| 1| 5| 4| 3| 2| 6| 0|

```

Example 4-14 Reverse using 3 instructions

```

/* Goal:  Reverse the order of the words */
/* Instruction Result */
      | 7| 6| 5| 4| 3| 2| 1| 0|

PSHUFLW (0,1,2,3)| 7| 6| 5| 4| 0| 1| 2| 3|

PSHUFHW (0,1,2,3)| 4| 5| 6| 7| 0| 1| 2| 3|

PSHUFD (1,0,3,2)| 0| 1| 2| 3| 4| 5| 6| 7|

```

Unpacking/interleaving 64-bit Data in 128-bit Registers

The `punpcklqdq/punpchkdq` instructions interleave the low/high-order 64-bits of the source operand and the low/high-order 64-bits of the destination operand and writes them to the destination register. The high/low-order 64-bits of the source operands are ignored.

Data Movement

There are two additional instructions to enable data movement from the 64-bit SIMD integer registers to the 128-bit SIMD registers.

The `movq2dq` instruction moves the 64-bit integer data from an MMX register (source) to a 128-bit destination register. The high-order 64 bits of the destination register are zeroed-out.

The `movdq2q` instruction moves the low-order 64-bits of integer data from a 128-bit source register to an MMX register (destination).

Conversion Instructions

New instructions have been added to support 4-wide conversion of single-precision data to/from double-word integer data. Also, conversions between double-precision data and double-word integer data have been added.

Generating Constants

The SIMD integer instruction sets do not have instructions that will load immediate constants to the SIMD registers. The following code segments generate frequently used constants in the SIMD register. Of course, you can also put constants as local variables in memory, but when doing so be sure to duplicate the values in memory and load the values with a `movq`, `movdqa`, or `movdqu` instructions, see [Example 4-15](#).

Example 4-15 Generating Constants

```
pxor    MM0, MM0    ; generate a zero register in MM0
pcmpeq  MM1, MM1    ; Generate all 1's in register MM1,
                    ; which is -1 in each of the packed
                    ; data type fields

pxor    MM0, MM0
pcmpeq  MM1, MM1
psubb   MM0, MM1 [psubw MM0, MM1] (psubd MM0, MM1)
```

continued

Example 4-15 Generating Constants (continued)

```

; three instructions above generate
; the constant 1 in every
; packed-byte [or packed-word]
; (or packed-dword) field

pcmpeq MM1, MM1
psrlw MM1, 16-n(psrlw MM1, 32-n)
; two instructions above generate
; the signed constant  $2^n-1$  in every
; packed-word (or packed-dword) field

pcmpeq MM1, MM1
psllw MM1, n (pslld MM1, n)
; two instructions above generate
; the signed constant  $-2^n$  in every
; packed-word (or packed-dword) field

```



NOTE. *Because the SIMD integer instruction sets do not support shift instructions for bytes, 2^n-1 and -2^n are relevant only for packed words and packed doublewords.*

Building Blocks

This section describes instructions and algorithms which implement common code building blocks efficiently.

Absolute Difference of Unsigned Numbers

[Example 4-16](#) computes the absolute difference of two unsigned numbers. It assumes an unsigned packed-byte data type. Here, we make use of the subtract instruction with unsigned saturation. This instruction receives UNSIGNED operands and subtracts them with UNSIGNED saturation. This support exists only for packed bytes and packed words, not for packed doublewords.

Example 4-16 Absolute Difference of Two Unsigned Numbers

```
; Input:
;      MM0 source operand
;      MM1 source operand
; Output:
;      MM0 absolute difference of the unsigned
;      operands

movq    MM2, MM0    ; make a copy of MM0
psubusb MM0, MM1    ; compute difference one way
psubusb MM1, MM2    ; compute difference the other way
por     MM0, MM1    ; OR them together
```

This example will not work if the operands are signed.

Note that the `psadbw` instruction may also be used in some situations; see section [“Packed Sum of Absolute Differences”](#) for details.

Absolute Difference of Signed Numbers

[Example 4-17](#) computes the absolute difference of two signed numbers.



NOTE. *There is no MMX™ technology subtract instruction that receives SIGNED operands and subtracts them with UNSIGNED saturation.*

The technique used here is to first sort the corresponding elements of the input operands into packed words of the maximum values, and packed words of the minimum values. Then the minimum values are subtracted from the maximum values to generate the required absolute difference. The key is a fast sorting technique that uses the fact that $B = \text{xor}(A, \text{xor}(A, B))$ and $A = \text{xor}(A, 0)$. Thus in a packed data type, having some elements being $\text{xor}(A, B)$ and some being 0, you could xor such an operand with A and receive in some places values of A and in some values of B . The following examples assume a packed-word data type, each element being a signed value.

Example 4-17 Absolute Difference of Signed Numbers

```

; Input :
;       MM0 signed source operand
;       MM1 signed source operand
; Output:
;       MM0 absolute difference of the unsigned
;       operands

movq    MM2, MM0    ; make a copy of source1 (A)
pcmpgtw MM0, MM1    ; create mask of
                    ; source1 > source2 (A > B)

movq    MM4, MM2    ; make another copy of A
pxor    MM2, MM1    ; create the intermediate value of
                    ; the swap operation - xor(A, B)

pand    MM2, MM0    ; create a mask of 0s and xor(A, B)
                    ; elements. Where A > B there will
                    ; be a value xor(A, B) and where
                    ; A <= B there will be 0.

pxor    MM4, MM2    ; minima - xor(A, swap mask)
pxor    MM1, MM2    ; maxima - xor(B, swap mask)
psubw   MM1, MM4    ; absolute difference =
                    ; maxima - minima

```

Absolute Value

Use [Example 4-18](#) to compute $|x|$, where x is signed. This example assumes signed words to be the operands.

Example 4-18 Computing Absolute Value

```

; Input:
;      MM0      signed source operand
; Output:
;      MM1      ABS(MM0)
pxor   MM1, MM1    ; set MM1 to all zeros
psubw  MM1, MM0    ; make each MM1 word contain the
                   ; negative of each MM0 word
pmaxsw MM1, MM0    ; MM1 will contain only the positive
                   ; (larger) values - the absolute value

```



CAUTION. *The absolute value of the most negative number (that is, 8000 hex for 16-bit) cannot be represented using positive numbers. This algorithm will return the original value for the absolute value (8000 hex).*

Clipping to an Arbitrary Range [high, low]

This section explains how to clip a values to a range [high, low]. Specifically, if the value is less than low or greater than high, then clip to low or high, respectively. This technique uses the packed-add and packed-subtract instructions with saturation (signed or unsigned), which means that this technique can only be used on packed-byte and packed-word data types.

The examples in this section use the constants `packed_max` and `packed_min` and show operations on word values. For simplicity we use the following constants (corresponding constants are used in case the operation is done on byte values):

- `packed_max` equals `0x7fff7fff7fff7fff`
- `packed_min` equals `0x8000800080008000`
- `packed_low` contains the value `low` in all four words of the packed-words data type
- `packed_high` contains the value `high` in all four words of the packed-words data type
- `packed_usmax` all values equal 1
- `high_us` adds the `high` value to all data elements (4 words) of `packed_min`
- `low_us` adds the `low` value to all data elements (4 words) of `packed_min`

Highly Efficient Clipping

For clipping signed words to an arbitrary range, the `pmaxsw` and `pminsw` instructions may be used. For clipping unsigned bytes to an arbitrary range, the `pmaxub` and `pminub` instructions may be used. [Example 4-19](#) shows how to clip signed words to an arbitrary range; the code for clipping unsigned bytes is similar.

Example 4-19 Clipping to a Signed Range of Words [high, low]

```
; Input:
;      MM0      signed source operands
; Output:
;      MM0      signed words clipped to the signed
;               range [high, low]

pminsw MM0, packed_high
pmaxsw MM0, packed_low
```

Example 4-20 Clipping to an Arbitrary Signed Range [high, low]

```

; Input:
;      MM0          signed source operands
; Output:
;      MM1          signed operands clipped to the unsigned
;                  range [high, low]

paddw  MM0, packed_min    ; add with no saturation
                        ; 0x8000 to convert to unsigned
paddusw MM0, (packed_usmax - high_us)
                        ; in effect this clips to high
psubusw MM0, (packed_usmax - high_us + low_us)
                        ; in effect this clips to low
paddw  MM0, packed_low    ; undo the previous two offsets

```

The code above converts values to unsigned numbers first and then clips them to an unsigned range. The last instruction converts the data back to signed data and places the data within the signed range. Conversion to unsigned data is required for correct results when $(\text{high} - \text{low}) < 0x8000$.

If $(\text{high} - \text{low}) \geq 0x8000$, the algorithm can be simplified as shown in [Example 4-21](#):

Example 4-21 Simplified Clipping to an Arbitrary Signed Range

```

; Input:  MM0          signed source operands
; Output: MM1          signed operands clipped to the unsigned
;                  range [high, low]

paddssw MM0, (packed_max - packed_high)
                        ; in effect this clips to high
psubssw MM0, (packed_usmax - packed_high + packed_low)
                        ; clips to low
paddw   MM0, low      ; undo the previous two offsets

```

This algorithm saves a cycle when it is known that $(\text{high} - \text{low}) \geq 0x8000$. The three-instruction algorithm does not work when $(\text{high} - \text{low}) < 0x8000$, because $0xffff$ minus any number $< 0x8000$ will yield a number greater in magnitude than $0x8000$, which is a negative number. When the second instruction, `psubssw MM0, (0xffff - high + low)`, in the three-step algorithm ([Example 4-21](#)) is executed, a negative number is subtracted. The result of this subtraction causes the values in `MM0` to be increased instead of decreased, as should be the case, and an incorrect answer is generated.

Clipping to an Arbitrary Unsigned Range [high, low]

The code in [Example 4-22](#) clips an unsigned value to the unsigned range `[high, low]`. If the value is less than `low` or greater than `high`, then clip to `low` or `high`, respectively. This technique uses the packed-add and packed-subtract instructions with unsigned saturation, thus this technique can only be used on packed-bytes and packed-words data types.

The example illustrates the operation on word values.

Example 4-22 Clipping to an Arbitrary Unsigned Range [high, low]

```

; Input:
;          MM0      unsigned source operands
; Output:
;          MM1      unsigned operands clipped to the unsigned
;                   range [HIGH, LOW]
paddusw    MM0, 0xffff - high
            ; in effect this clips to high
psubusw    MM0, (0xffff - high + low)
            ; in effect this clips to low
paddw      MM0, low
            ; undo the previous two offsets

```

Packed Max/Min of Signed Word and Unsigned Byte

Signed Word

The `pmaxsw` instruction returns the maximum between the four signed words in either two SIMD registers, or one SIMD register and a memory location.

The `pminsw` instruction returns the minimum between the four signed words in either two SIMD registers, or one SIMD register and a memory location.

Unsigned Byte

The `pmaxub` instruction returns the maximum between the eight unsigned bytes in either two SIMD registers, or one SIMD register and a memory location.

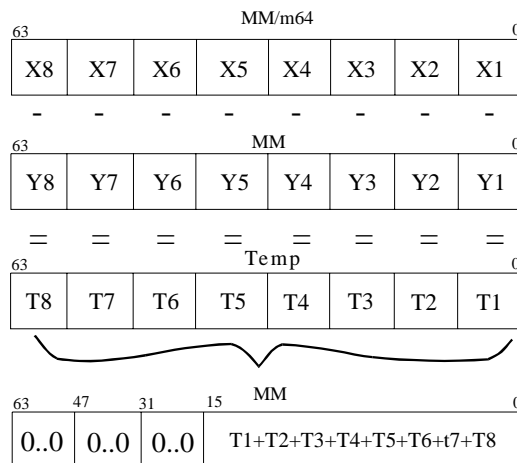
The `pminub` instruction returns the minimum between the eight unsigned bytes in either two SIMD registers, or one SIMD register and a memory location.

Packed Multiply High Unsigned

The `pmulhuw` and `pmulhw` instruction multiplies the unsigned/signed words in the destination operand with the unsigned/signed words in the source operand. The high-order 16 bits of the 32-bit intermediate results are written to the destination operand.

Packed Sum of Absolute Differences

The `psadbw` instruction (see [Figure 4-9](#)) computes the absolute value of the difference of unsigned bytes for either two SIMD registers, or one SIMD register and a memory location. These differences are then summed to produce a word result in the lower 16-bit field, and the upper three words are set to zero.

Figure 4-9 PSADBW Instruction Example

The subtraction operation presented above is an absolute difference, that is, $t = \text{abs}(x - y)$. The byte values are stored in temporary space, all values are summed together, and the result is written into the lower word of the destination register.

Packed Average (Byte/Word)

The `pavgb` and `pavgw` instructions add the unsigned data elements of the source operand to the unsigned data elements of the destination register, along with a carry-in. The results of the addition are then each independently shifted to the right by one bit position. The high order bits of each element are filled with the carry bits of the corresponding sum.

The destination operand is an SIMD register. The source operand can either be an SIMD register or a memory operand.

The `PAVGB` instruction operates on packed unsigned bytes and the `PAVGW` instruction operates on packed unsigned words.

Complex Multiply by a Constant

Complex multiplication is an operation which requires four multiplications and two additions. This is exactly how the `pmaddwd` instruction operates. In order to use this instruction, you need to format the data into multiple 16-bit values. The real and imaginary components should be 16-bits each. Consider [Example 4-23](#), which assumes that the 64-bit MMX registers are being used:

- Let the input data be D_r and D_i where D_r is real component of the data and D_i is imaginary component of the data.
- Format the constant complex coefficients in memory as four 16-bit values [C_r $-C_i$ C_i C_r]. Remember to load the values into the MMX register using a `movq` instruction.
- The real component of the complex product is

$$P_r = D_r * C_r - D_i * C_i$$
and the imaginary component of the complex product is $P_i = D_r * C_i + D_i * C_r$.

Example 4-23 Complex Multiply by a Constant

```

; Input:
;          MM0          complex value, Dr, Di
;          MM1          constant complex coefficient in the form
;                      [Cr -Ci Ci Cr]
; Output:
;          MM0          two 32-bit dwords containing [Pr Pi]
;
punpckldq  MM0, MM0      ; makes [Dr Di Dr Di]
pmaddwd    MM0, MM1      ; done, the result is
                        ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]

```

Note that the output is a packed doubleword. If needed, a `pack` instruction can be used to convert the result to 16-bit (thereby matching the format of the input).

Packed 32*32 Multiply

The `PMULUDQ` instruction performs an unsigned multiply on the lower pair of double-word operands within each 64-bit chunk from the two sources; the full 64-bit result from each multiplication is returned to the destination register. This instruction is added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

Packed 64-bit Add/Subtract

The `PADDQ/PSUBQ` instructions add/subtract quad-word operands within each 64-bit chunk from the two sources; the 64-bit result from each computation is written to the destination register. Like the integer `ADD/SUB` instruction, `PADDQ/PSUBQ` can operate on either unsigned or signed (two's complement notation) integer operands. When an individual result is too large to be represented in 64-bits, the lower 64-bits of the result are written to the destination operand and therefore the result wraps around. These instructions are added in both a 64-bit and 128-bit version; the latter performs 2 independent operations, on the low and high halves of a 128-bit register.

128-bit Shifts

The `PSLLDQ/PSRLDQ` instructions shift the first operand to the left/right by the amount of bytes specified by the immediate operand. The empty low/high-order bytes are cleared (set to zero). If the value specified by the immediate operand is greater than 15, then the destination is set to all zeros.

Memory Optimizations

You can improve memory accesses using the following techniques:

- Avoiding partial memory accesses
- Increasing the bandwidth of memory fills and video fills
- Prefetching data with Streaming SIMD Extensions (see [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#)).

The MMX registers and XMM registers allow you to move large quantities of data without stalling the processor. Instead of loading single array values that are 8, 16, or 32 bits long, consider loading the values in a single quadword or double quadword, then incrementing the structure or array pointer accordingly.

Any data that will be manipulated by SIMD integer instructions should be loaded using either:

- the SIMD integer instruction that loads a 64-bit or 128-bit operand (for example, `movq MM0, m64`)
- the register-memory form of any SIMD integer instruction that operates on a quadword or double quadword memory operand (for example, `pmaddw MM0, m64`).

All SIMD data should be stored using the SIMD integer instruction that stores a 64-bit or 128-bit operand (for example, `movq m64, MM0`)

The goal of these recommendations is twofold. First, the loading and storing of SIMD data is more efficient using the larger block sizes. Second, this helps to avoid the mixing of 8-, 16-, or 32-bit load and store operations with SIMD integer technology load and store operations to the same SIMD data. This, in turn, prevents situations in which small loads follow large stores to the same area of memory, or large loads follow small stores to the same area of memory. The Pentium II, Pentium III, and Pentium 4 processors stall in these situations; see [Chapter 2, “General Optimization Guidelines”](#) for more details.

Partial Memory Accesses

Consider a case with large load after a series of small stores to the same area of memory (beginning at memory address `mem`). The large load will stall in this case as shown in [Example 4-24](#).

Example 4-24 A Large Load after a Series of Small Stores (Penalty)

```

mov    mem, eax        ; store dword to address "mem"
mov    mem + 4, ebx    ; store dword to address "mem + 4"
      :
      :
movq   mm0, mem        ; load qword at address "mem", stalls

```

The `movq` must wait for the stores to write memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory). When you change the code sequence as shown in [Example 4-25](#), the processor can access the data without delay.

Example 4-25 Accessing Data without Delay

```

movd    mm1, ebx        ; build data into a qword first
                               ; before storing it to memory

movd    mm2, eax
psllq   mm1, 32
por     mm1, mm2
movq    mem, mm1        ; store SIMD variable to "mem" as
                               ; a qword
:
:
movq    mm0, mem        ; load qword SIMD "mem", no stall

```

Let us now consider a case with a series of small loads after a large store to the same area of memory (beginning at memory address `mem`) as shown in [Example 4-26](#). Most of the small loads will stall because they are not aligned with the store; see “Store Forwarding” in Chapter 2 for more details.

Example 4-26 A Series of Small Loads after a Large Store

```

movq    mem, mm0        ; store qword to address "mem"
:
:
mov     bx, mem + 2      ; load word at "mem + 2" stalls
mov     cx, mem + 4      ; load word at "mem + 4" stalls

```

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). When you change the code sequence as shown in [Example 4-27](#), the processor can access the data without delay.

Example 4-27 Eliminating Delay for a Series of Small Loads after a Large Store

```
movq    mem, mm0    ; store qword to address "mem"
:
:
movq    mm1, mem     ; load qword at address "mem"
movd    eax, mm1     ; transfer "mem + 2" to eax from
                    ; MMX register, not memory
psrlq   mm1, 32
shr     eax, 16
movd    ebx, mm1     ; transfer "mem + 4" to bx from
                    ; MMX register, not memory
and     ebx, 0ffffh
```

These transformations, in general, increase the number of instructions required to perform the desired operation. For Pentium II, Pentium III, and Pentium 4 processors, the benefit of avoiding forwarding problems outweighs the performance penalty due to the increased number of instructions, making the transformations worthwhile.

Increasing Bandwidth of Memory Fills and Video Fills

It is beneficial to understand how memory is accessed and filled. A memory-to-memory fill (for example a memory-to-video fill) is defined as a 64-byte (cache line) load from memory which is immediately stored back to memory (such as a video frame buffer). The following are guidelines for obtaining higher bandwidth and shorter latencies for sequential memory fills (video fills). These recommendations are relevant for all Intel architecture processors with MMX technology and refer to cases in which the loads and stores do not hit in the first- or second-level cache.

Increasing Memory Bandwidth Using the MOVDQ Instruction

Loading any size data operand will cause an entire cache line to be loaded into the cache hierarchy. Thus any size load looks more or less the same from a memory bandwidth perspective. However, using many smaller loads consumes more microarchitectural resources than fewer larger stores. Consuming too many of these resources can cause the processor to stall and reduce the bandwidth that the processor can request of the memory subsystem.

Using `movdq` to store the data back to UC memory (or WC memory in some cases) instead of using 32-bit stores (for example, `movd`) will reduce by three-quarters the number of stores per memory fill cycle. As a result, using the `movdq` instruction in memory fill cycles can achieve significantly higher effective bandwidth than using the `movd` instruction.

Increasing Memory Bandwidth by Loading and Storing to and from the Same DRAM Page

DRAM is divided into pages, which are not the same as operating system (OS) pages. The size of a DRAM page is a function of the total size of the DRAM and the organization of the DRAM. Page sizes of several Kilobytes are common. Like OS pages, DRAM pages are constructed of sequential addresses. Sequential memory accesses to the same DRAM page have shorter latencies than sequential accesses to different DRAM pages. In many systems the latency for a page miss (that is, an access to a different page instead of the page previously accessed) can be twice as large as the latency of a memory page hit (access to the same page as the previous access). Therefore, if the loads and stores of the memory fill cycle are to the same DRAM page, a significant increase in the bandwidth of the memory fill cycles can be achieved.

Increasing UC and WC Store Bandwidth by Using Aligned Stores

Using aligned stores to fill UC or WC memory will yield higher bandwidth than using unaligned stores. If a UC store or some WC stores cross a cache line boundary, a single store will result in two transaction on the bus, reducing the efficiency of the bus transactions. By aligning the stores to the size of the stores, you eliminate the possibility of crossing a cache line boundary, and the stores will not be split into separate transactions.

Converting from 64-bit to 128-bit SIMD Integer

The SSE2 define a superset of 128-bit integer instructions currently available in MMX technology; the operation of the extended instructions remains the same and simply operate on data that is twice as wide. This simplifies porting of current 64-bit integer applications. However, there are few additional considerations:

- Computation instructions which use a memory operand that may not be aligned to a 16-byte boundary must be replaced with an unaligned 128-bit load (`movdqu`) followed by the same computation operation that uses instead register operands. Use of 128-bit integer computation instructions with memory operands that are not 16-byte aligned will result in a General Protection fault. The unaligned 128-bit load and store is not as efficient as the corresponding aligned versions; this can reduce the performance gains when using the 128-bit SIMD integer extensions. The general guidelines on the alignment of memory operands are:
 - The greatest performance gains can be achieved when all memory streams are 16-byte aligned.
 - Reasonable performance gains are possible if roughly half of all memory streams are 16-byte aligned, and the other half are not.
 - Little or no performance gain may result if all memory streams are not aligned to 16-bytes; in this case, use of the 64-bit SIMD integer instructions may be preferable.
- Loop counters need to be updated because each 128-bit integer instruction operates on twice the amount of data as the 64-bit integer counterpart.
- Extension of the `pshufw` instruction (shuffle word across 64-bit integer operand) across a full 128-bit operand is emulated by a combination of the following instructions: `pshufhw`, `pshufwlw`, `pshufd`.
- Use of the 64-bit shift by bit instructions (`psrlq`, `psllq`) are extended to 128 bits in these ways:
 - use of `psrlq` and `psllq`, along with masking logic operations
 - code sequence is rewritten to use the `psrldq` and `pslldq` instructions (shift double quad-word operand by bytes).

Optimizing for SIMD Floating-point Applications

5

This chapter discusses general rules of optimizing for the single-instruction, multiple-data (SIMD) floating-point instructions available in Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2). This chapter also provides examples that illustrate the optimization techniques for single-precision and double-precision SIMD floating-point applications.

General Rules for SIMD Floating-point Code

The rules and suggestions listed in this section help optimize floating-point code containing SIMD floating-point instructions. Generally, it is important to understand and balance port utilization to create efficient SIMD floating-point code. The basic rules and suggestions include the following:

- Follow all guidelines in [Chapter 2](#) and [Chapter 3](#).
- Exceptions: mask exceptions to achieve higher performance. When exceptions are unmasked, software performance is slower.
- Utilize the flush-to-zero mode for higher performance to avoid the penalty of dealing with denormals and underflows.
- Incorporate the prefetch instruction whenever possible (for details, refer to [Chapter 6, “Optimizing Cache Usage for Intel Pentium 4 Processors”](#)).
- Use MMX technology instructions and registers if the computations can be done in SIMD integer for shuffling data.
- Use MMX technology instructions and registers or for copying data that is not used later in SIMD floating-point computations.

- Use the reciprocal instructions followed by iteration for increased accuracy. These instructions yield reduced accuracy but execute much faster. Note the following:
 - If reduced accuracy is acceptable, use them with no iteration.
 - If near full accuracy is needed, use a Newton-Raphson iteration.
 - If full accuracy is needed, then use divide and square root which provide more accuracy, but slow down performance.

Planning Considerations

Whether adapting an existing application or creating a new one, using SIMD floating-point instructions to achieve optimum performance gain requires programmers to consider several issues. In general, when choosing candidates for optimization, look for code segments that are computationally intensive and floating-point intensive. Also consider efficient use of the cache architecture.

The sections that follow answer the questions that should be raised before implementation:

- Which part of the code benefits from SIMD floating-point instructions?
- Is the current algorithm the most appropriate for SIMD floating-point instructions?
- Is the code floating-point intensive?
- Do either single-precision floating-point or double-precision floating-point computations provide enough range and precision?
- Is the data arranged for efficient utilization of the SIMD floating-point registers?
- Is this application targeted for processors without SIMD floating-point instructions?

For more details, see the section on “[Considerations for Code Conversion to SIMD Programming](#)” in Chapter 3.

Detecting SIMD Floating-point Support

Applications must be able to determine if SSE are available. Please refer the section “[Checking for Processor Support of SIMD Technologies](#)” in Chapter 3 for the techniques to determine whether the processor and operating system support SSE.

Using SIMD Floating-point with x87 Floating-point

Because the XMM registers used for SIMD floating-point computations are separate registers and are not mapped onto the existing x87 floating-point stack, SIMD floating-point code can be mixed with either x87 floating-point or 64-bit SIMD integer code.

Scalar Floating-point Code

There are SIMD floating-point instructions that operate only on the least-significant operand in the SIMD register. These instructions are known as scalar instructions. They allow the XMM registers to be used for general-purpose floating-point computations.

In terms of performance, scalar floating-point code can be equivalent to or exceed x87 floating-point code, and has the following advantages:

- SIMD floating-point code uses a flat register model, whereas x87 floating-point code uses a stack model. Using scalar floating-point code eliminates the need to use `fxch` instructions, which has some performance limit on the Intel Pentium 4 processor.
- Mixing with MMX technology code without penalty.
- Flush-to-zero mode.
- Shorter latencies than x87 floating-point.

When using scalar floating-point instructions, it is not necessary to ensure that the data appears in vector form. However, all of the optimizations regarding alignment, scheduling, instruction selection, and other optimizations covered in Chapters 2 and 3 should be observed.

Data Alignment

SIMD floating-point data is 16-byte aligned. Referencing unaligned 128-bit SIMD floating-point data will result in an exception unless the `movups` or `movupd` (move unaligned packed single or unaligned packed double) instruction is used. The unaligned instructions used on aligned or unaligned data will also suffer a performance penalty relative to aligned accesses.

Refer to section “[Stack and Data Alignment](#)” in Chapter 3 for more information.

Data Arrangement

Because the SSE and SSE2 incorporate a SIMD architecture, arranging the data to fully use the SIMD registers produces optimum performance. This implies contiguous data for processing, which leads to fewer cache misses and can potentially quadruple the data throughput when using SSE, or twice the throughput when using SSE2. These performance gains can occur because four data element can be loaded with 128-bit load instructions into XMM registers using SSE (`movaps` – move aligned packed single precision). Similarly, two data element can loaded with 128-bit load instructions into XMM registers using SSE2 (`movapd` – move aligned packed double precision).

Refer to the “[Stack and Data Alignment](#)” in Chapter 3 for data arrangement recommendations. Duplicating and padding techniques overcome the misalignment problem that can occur in some data structures and arrangements. This increases the data space but avoids the expensive penalty for misaligned data access.

For some applications, the traditional data arrangement requires some changes to fully utilize the SIMD registers and parallel techniques. Traditionally, the data layout has been an array of structures (AoS). To fully utilize the SIMD registers, a new data layout has been proposed—a structure of arrays (SoA) resulting in more optimized performance.

Vertical versus Horizontal Computation

Traditionally, the AoS data structure is used in 3D geometry computations. SIMD technology can be applied to AoS data structure using a horizontal computation technique. This means that the *x*, *y*, *z*, and *w* components of a single vertex structure (that is, of a single vector simultaneously referred to as an *xyz* data representation, see the diagram below) are computed in parallel, and the array is updated one vertex at a time.

X	Y	Z	W
---	---	---	---

To optimally utilize the SIMD registers, the data structure can be organized in the SoA format. The SoA data structure enables a vertical computation technique, and is recommended over horizontal computation, for the following reasons:

- When computing on a single vector (xyz), it is common to use only a subset of the vector components; for example, in 3D graphics the w component is sometimes ignored. This means that for single-vector operations, 1 of 4 computation slots is not being utilized. This typically results in a 25% reduction of peak efficiency.
- It may become difficult to hide long latency operations. For instance, another common function in 3D graphics is normalization, which requires the computation of a reciprocal square root (that is, $1/\text{sqrt}$). Both the division and square root are long latency operations. With vertical computation (SoA), each of the 4 computation slots in a SIMD operation is producing a unique result, so the net latency per slot is $L/4$ where L is the overall latency of the operation. However, for horizontal computation, the 4 computation slots each produce the same result, hence to produce 4 separate results requires a net latency per slot of L .

To utilize all 4 computation slots, the vertex data can be reorganized to allow computation on each component of 4 separate vertices, that is, processing multiple vectors simultaneously. This can also be referred to as an SoA form of representing vertices data shown in [Table 5-1](#).

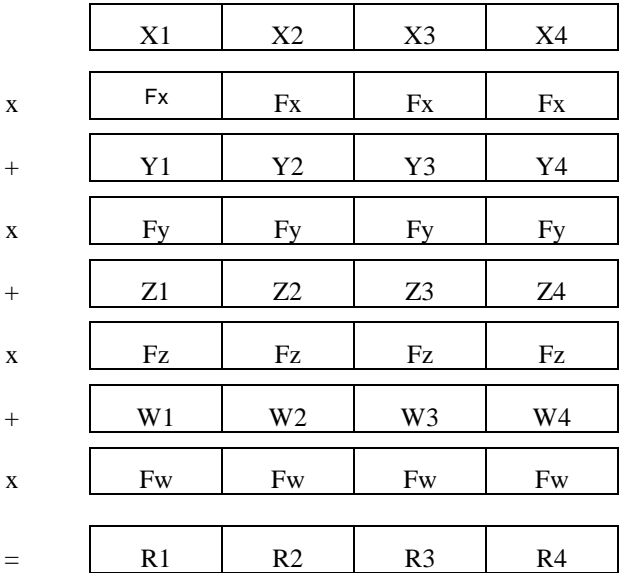
Table 5-1 SoA Form of Representing Vertices Data

Vx array	X1	X2	X3	X4	Xn
Vy array	Y1	Y2	Y3	Y4	Yn
Vz array	Z1	Z2	Z3	Y4	Zn
Vw array	W1	W2	W3	W4	Wn

Organizing data in this manner yields a unique result for each computational slot for each arithmetic operation.

Vertical computation takes advantage of the inherent parallelism in 3D geometry processing of vertices. It assigns the computation of four vertices to the four compute slots of the Pentium III processor, thereby eliminating the disadvantages of the horizontal approach described earlier. The dot product operation implements the SoA representation of vertices data. A schematic representation of dot product operation is shown in [Figure 5-1](#).

Figure 5-1 Dot Product Operation



[Figure 5-1](#) shows how 1 result would be computed for 7 instructions if the data were organized as AoS: 4 results would require 28 instructions.

Example 5-1 Pseudocode for Horizontal (xyz, AoS) Computation

```
mulps      ; x*x', y*y', z*z'
movaps     ; reg->reg move, since next steps overwrite
shufps     ; get b,a,d,c from a,b,c,d
addps      ; get a+b,a+b,c+d,c+d
movaps     ; reg->reg move
shufps     ; get c+d,c+d,a+b,a+b from prior addps
addps      ; get a+b+c+d,a+b+c+d,a+b+c+d,a+b+c+d
```

Now consider the case when the data is organized as SoA. [Example 5-2](#) demonstrates how 4 results are computed for 5 instructions.

Example 5-2 Pseudocode for Vertical (xxxx, yyyy, zzzz, SoA) Computation

```
mulps    ; x*x' for all 4 x-components of 4 vertices
mulps    ; y*y' for all 4 y-components of 4 vertices
mulps    ; z*z' for all 4 z-components of 4 vertices
addps    ; x*x' + y*y'
addps    ; x*x' + y*y' + z*z'
```

For the most efficient use of the four component-wide registers, reorganizing the data into the SoA format yields increased throughput and hence much better performance for the instructions used.

As can be seen from this simple example, vertical computation yielded 100% use of the available SIMD registers and produced 4 results. (The results may vary based on the application.) If the data structures must be in a format that is not “friendly” to vertical computation, it can be rearranged “on the fly” to achieve full utilization of the SIMD registers. This operation is referred to as “swizzling” operation and the reverse operation is referred to as “deswizzling.”

Data Swizzling

Swizzling data from one format to another is required in many algorithms. An example of this is AoS format, where the vertices come as *xyz* adjacent coordinates.

Rearranging them into SoA format, *xxxx*, *yyyy*, *zzzz*, allows more efficient SIMD computations. For efficient data shuffling and swizzling use the following instructions:

- `movlps`, `movhps` load/store and move data on half sections of the registers
- `shufps`, `unpackhps`, and `unpacklps` unpack data

To gather data from 4 different memory locations on the fly, follow steps:

1. Identify the first half of the 128-bit memory location.
2. Group the different halves together using the `movlps` and `movhps` to form an *xyxy* layout in two registers
3. From the 4 attached halves, get the *xxxx* by using one shuffle, the *yyyy* by using another shuffle.

The zzzz is derived the same way but only requires one shuffle.

[Example 5-3](#) illustrates the swizzle function.

Example 5-3 Swizzling Data

```
typedef struct _VERTEX_AOS {
    float x, y, z, color;
} Vertex_aos;                // AoS structure declaration
typedef struct _VERTEX_SOA {
    float x[4], float y[4], float z[4];
    float color[4];
} Vertex_soa;                // SoA structure declaration
void swizzle_asm (Vertex_aos *in, Vertex_soa *out)
{
    // in mem: x1y1z1w1-x2y2z2w2-x3y3z3w3-x4y4z4w4-
    // SWIZZLE XYZW --> XXXX
    asm {
        mov ecx, in           // get structure addresses
        mov edx, out

    y1 x1
        movhps xmm7, [ecx+16]  // xmm7 = y2 x2 y1 x1
        movlps xmm0, [ecx+32]  // xmm0 = -- -- y3 x3
        movhps xmm0, [ecx+48]  // xmm0 = y4 x4 y3 x3
        movaps xmm6, xmm7      // xmm6 = y1 x1 y1 x1
        shufps xmm7, xmm0, 0x88 // xmm7 = x1 x2 x3 x4 => X
        shufps xmm6, xmm0, 0xDD // xmm6 = y1 y2 y3 y4 => Y
        movlps xmm2, [ecx+8]   // xmm2 = -- -- w1 z1
        movhps xmm2, [ecx+24]  // xmm2 = w2 z2 u1 z1
        movlps xmm1, [ecx+40]  // xmm1 = -- -- s3 z3
        movhps xmm1, [ecx+56]  // xmm1 = w4 z4 w3 z3
```

continued

Example 5-3 Swizzling Data (continued)

```

        movaps xmm0, xmm2           // xmm0 = w1 z1 w1 z1
        shufps xmm2, xmm1, 0x88     // xmm2 = z1 z2 z3 z4 => Z
        movlps xmm7, [ecx]          // xmm7 = -- --shufps xmm0, xmm1,
                                   // 0xDD xmm6 = w1 w2 w3 w4 => W

        movaps [edx], xmm7          // store X
        movaps [edx+16], xmm6       // store Y
        movaps [edx+32], xmm2       // store Z
        movaps [edx+48], xmm0       // store W
                                   // SWIZZLE XYZ -> XXX
    }
}

```

[Example 5-4](#) shows the same data -swizzling algorithm encoded using the Intel® C++ Compiler's intrinsics for SSE.

Example 5-4 Swizzling Data Using Intrinsics

```

//Intrinsics version of data swizzle
void swizzle_intrin (Vertex_aos *in, Vertex_soa *out, int stride)
{
    __m128 x, y, z, w;
    __m128 tmp;
    x = _mm_loadl_pi(x, (__m64 *) (in));
    x = _mm_loadh_pi(x, (__m64 *) (stride + (char *) (in)));
    y = _mm_loadl_pi(y, (__m64 *) (2*stride+(char *) (in)));
    y = _mm_loadh_pi(y, (__m64 *) (3*stride+(char *) (in)));
    tmp = _mm_shuffle_ps( x, y, _MM_SHUFFLE( 2, 0, 2, 0));
    y = _mm_shuffle_ps( x, y, _MM_SHUFFLE( 3, 1, 3, 1));
    x = tmp;
}

```

continued

Example 5-4 Swizzling Data Using Intrinsics (continued)

```

z = _mm_loadl_pi(z, (__m64 *) (8 + (char *) (in)));
z = _mm_loadh_pi(z, (__m64 *) (stride+8+(char *) (in)));
w = _mm_loadl_pi(w, (__m64 *) (2*stride+8+(char *) (in)));
w = _mm_loadh_pi(w, (__m64 *) (3*stride+8+(char *) (in)));
tmp = _mm_shuffle_ps( z, w, _MM_SHUFFLE( 2, 0, 2, 0));
w = _mm_shuffle_ps( z, w, _MM_SHUFFLE( 3, 1, 3, 1));
z = tmp;
_mm_store_ps(&out->x[0], x);
_mm_store_ps(&out->y[0], y);
_mm_store_ps(&out->z[0], z);
_mm_store_ps(&out->w[0], w);
}

```



CAUTION. *Avoid creating a dependence chain from previous computations because the `movhps/movlps` instructions bypass one part of the register. The same issue can occur with the use of an exclusive-OR function within an inner loop in order to clear a register:*

```

xorps xmm0, xmm0 ; All 0's written to xmm0

```

Although the generated result of all zeros does not depend on the specific data contained in the source operand (that is, XOR of a register with itself always produces all zeros), the instruction cannot execute until the instruction that generates `xmm0` has completed. In the worst case, this creates a dependence chain that links successive iterations of the loop, even if those iterations are otherwise independent. The performance impact can be significant depending on how many other independent intra-loop computations are performed. Note that on the Pentium 4 processor, the SIMD integer `pxor` instructions, if used with the same register, do break the dependence chain, eliminating false dependencies when clearing registers.

The same situation can occur for the above `movhps/movlps/shufps` sequence. Since each `movhps/movlps` instruction bypasses part of the destination register, the instruction cannot execute until the prior instruction that generates this register has completed. As with the `xorps` example, in the worst case this dependence can prevent successive loop iterations from executing in parallel.

A solution is to include a 128-bit load (that is, from a dummy local variable, such as `tmp` in [Example 5-4](#)) to each register to be used with a `movhps/movlps` instruction. This action effectively breaks the dependence by performing an independent load from a memory or cached location.

Data Deswizzling

In the deswizzle operation, we want to arrange the SoA format back into AoS format so the `xxxx, yyyy, zzzz` are rearranged and stored in memory as `xyz`. To do this we can use the `unpcklps/unpckhps` instructions to regenerate the `xyxy` layout and then store each half (`xy`) into its corresponding memory location using `movlps/movhps` followed by another `movlps/movhps` to store the `z` component.

[Example 5-5](#) illustrates the deswizzle function:

Example 5-5 Deswizzling Single-Precision SIMD Data

```
void deswizzle_asm(Vertex_soa *in, Vertex_aos *out)
{
    __asm {
        mov     ecx, in           // load structure addresses
        mov     edx, out
        movaps  xmm7, [ecx]       // load x1 x2 x3 x4 => xmm7
        movaps  xmm6, [ecx+16]    // load y1 y2 y3 y4 => xmm6
        movaps  xmm5, [ecx+32]    // load z1 z2 z3 z4 => xmm5
        movaps  xmm4, [ecx+48]    // load w1 w2 w3 w4 => xmm4
```

continued

Example 5-5 Deswizzling Single-Precision SIMD Data (continued)

```

// START THE DESWIZZLING HERE
    movaps    xmm0, xmm7           // xmm0= x1 x2 x3 x4
    unpcklps  xmm7, xmm6           // xmm7= x1 y1 x2 y2
    movlps    [edx], xmm7          // v1 = x1 y1 -- --
    movhps    [edx+16], xmm7       // v2 = x2 y2 -- --
    unpckhps  xmm0, xmm6           // xmm0= x3 y3 x4 y4
    movlps    [edx+32], xmm0       // v3 = x3 y3 -- --
    movhps    [edx+48], xmm0       // v4 = x4 y4 -- --
    movaps    xmm0, xmm5           // xmm0= z1 z2 z3 z4
    unpcklps  xmm5, xmm4           // xmm5= z1 w1 z2 w2
    unpckhps  xmm0, xmm4           // xmm0= z3 w3 z4 w4
    movlps    [edx+8], xmm5        // v1 = x1 y1 z1 w1
    movhps    [edx+24], xmm5       // v2 = x2 y2 z2 w2
    movlps    [edx+40], xmm0       // v3 = x3 y3 z3 w3
    movhps    [edx+56], xmm0       // v4 = x4 y4 z4 w4
// DESWIZZLING ENDS HERE
    }
}

```

You may have to swizzle data in the registers, but not in memory. This occurs when two different functions need to process the data in different layout. In lighting, for example, data comes as `rrrr gggg bbbb aaaa`, and you must deswizzle them into `rgba` before converting into integers. In this case you use the `movlhps/movhlps` instructions to do the first part of the deswizzle followed by `shuffle` instructions, see [Example 5-6](#) and [Example 5-7](#).

Example 5-6 Deswizzling Data Using the movhlps and shuffle Instructions

```

void deswizzle_rgb(Vertex_soa *in, Vertex_aos *out)
{
    //---deswizzle rgb---
    // assume: xmm1=rrrr, xmm2=gggg, xmm3=bbbb, xmm4=aaaa
    __asm {
        mov     ecx, in           // load structure addresses
        mov     edx, out
        movaps  xmm1, [ecx]       // load r1 r2 r3 r4 => xmm1
        movaps  xmm2, [ecx+16]    // load g1 g2 g3 g4 => xmm2
        movaps  xmm3, [ecx+32]    // load b1 b2 b3 b4 => xmm3
        movaps  xmm4, [ecx+48]    // load a1 a2 a3 a4 => xmm4

        // Start deswizzling here
        movaps  xmm7, xmm4        // xmm7= a1 a2 a3 a4
        movhlps xmm7, xmm3        // xmm7= b3 b4 a3 a4
        movaps  xmm6, xmm2        // xmm6= g1 g2 g3 g4
        movhlps xmm3, xmm4        // xmm3= b1 b2 a1 a2
        movhlps xmm2, xmm1        // xmm2= r3 r4 g3 g4
        movhlps xmm1, xmm6        // xmm1= r1 r2 g1 g2
        movaps  xmm6, xmm2        // xmm6= r3 r4 g3 g4
        movaps  xmm5, xmm1        // xmm5= r1 r2 g1 g2
        shufps  xmm2, xmm7, 0xDD  // xmm2= r4 g4 b4 a4
        shufps  xmm1, xmm3, 0x88  // xmm4= r1 g1 b1 a1
        shufps  xmm5, xmm3, 0x88  // xmm5= r2 g2 b2 a2
        shufps  xmm6, xmm7, 0xDD  // xmm6= r3 g3 b3 a3
        movaps  [edx], xmm4       // v1 = r1 g1 b1 a1
        movaps  [edx+16], xmm5    // v2 = r2 g2 b2 a2
        movaps  [edx+32], xmm6    // v3 = r3 g3 b3 a3
    }
}

```

continued

Example 5-6 Deswizzling Data Using the movltps and shuffle Instructions (continued)

```

movaps [edx+48], xmm2          // v4 = r4 g4 b4 a4
// DESWIZZLING ENDS HERE
    }
}

```

Example 5-7 Deswizzling Data 64-bit Integer SIMD Data

```

void mmx_deswizzle(IVertex_soa *in, IVertex_aos *out)
{
    __asm {
        mov     ebx, in
        mov     edx, out
        movq    mm0, [ebx]        // mm0= u1 u2
        movq    mm1, [ebx+16]     // mm1= v1 v2
        movq    mm2, mm0         // mm2= u1 u2
        punpckhdq mm0, mm1       // mm0= u1 v1
        punpckldq mm2, mm1       // mm0= u2 v2
        movq    [edx], mm2       // store u1 v1
        movq    [edx+8], mm0     // store u2 v2
        movq    mm4, [ebx+8]     // mm0= u3 u4
        movq    mm5, [ebx+24]    // mm1= v3 v4
        movq    mm6, mm4         // mm2= u3 u4
        punpckhdq mm4, mm5       // mm0= u3 v3
        punpckldq mm6, mm5       // mm0= u4 v4
        movq    [edx+16], mm6    // store u3v3
        movq    [edx+24], mm4    // store u4v4
    }
}

```

Using MMX Technology Code for Copy or Shuffling Functions

If there are some parts in the code that are mainly copying, shuffling, or doing logical manipulations that do not require use of SSE code, consider performing these actions with MMX technology code. For example, if texture data is stored in memory as SoA (uuuu, vvvv) and they need only to be deswizzled into AoS layout (uv) for the graphic cards to process, you can use either the SSE or MMX technology code. Using the MMX instructions allow you to conserve XMM registers for other computational tasks.

[Example 5-8](#) illustrates how to use MMX technology code for copying or shuffling.

Example 5-8 Using MMX Technology Code for Copying or Shuffling

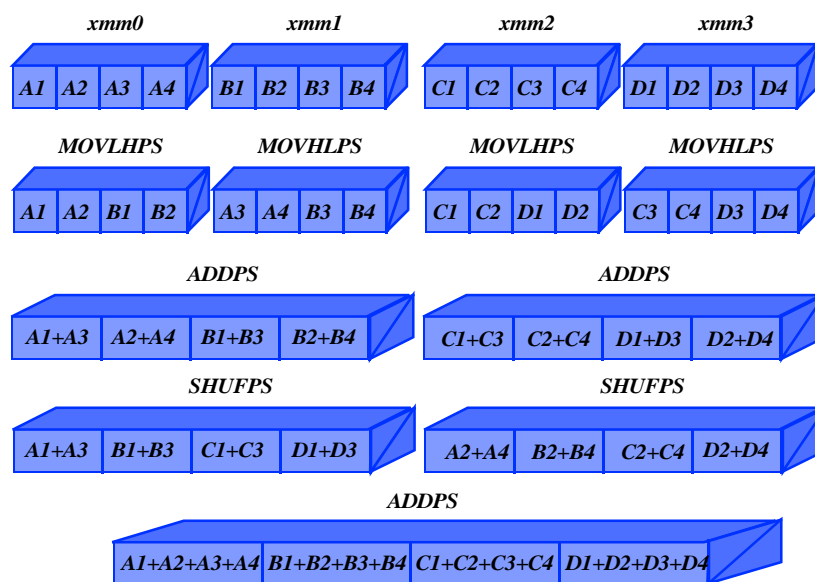
<code>movq</code>	<code>mm0, [Uarray+ebx]</code>	<code>; mm0= u1 u2</code>
<code>movq</code>	<code>mm1, [Varray+ebx]</code>	<code>; mm1= v1 v2</code>
<code>movq</code>	<code>mm2, mm0</code>	<code>; mm2= u1 u2</code>
<code>punpckhdq</code>	<code>mm0, mm1</code>	<code>; mm0= u1 v1</code>
<code>punpckldq</code>	<code>mm2, mm1</code>	<code>; mm2= u2 v2</code>
<code>movq</code>	<code>[Coords+edx], mm0</code>	<code>; store u1 v1</code>
<code>movq</code>	<code>[Coords+8+edx], mm2</code>	<code>; store u2 v2</code>
<code>movq</code>	<code>mm4, [Uarray+8+ebx]</code>	<code>; mm4= u3 u4</code>
<code>movq</code>	<code>mm5, [Varray+8+ebx]</code>	<code>; mm5= v3 v4</code>
<code>movq</code>	<code>mm6, mm4</code>	<code>; mm6= u3 u4</code>
<code>punpckhdq</code>	<code>mm4, mm5</code>	<code>; mm4= u3 v3</code>
<code>punpckldq</code>	<code>mm6, mm5</code>	<code>; mm6= u4 v4</code>
<code>movq</code>	<code>[Coords+16+edx], mm4</code>	<code>; store u3 v3</code>
<code>movq</code>	<code>[Coords+24+edx], mm6</code>	<code>; store u4 v4</code>

Horizontal ADD

Although vertical computations use the SIMD performance better than horizontal computations do, in some cases, the code must use a horizontal operation. The `movlhps/movhlps` and `shuffle` can be used to sum data horizontally. For example, starting with four 128-bit registers, to sum up each register horizontally while having

the final results in one register, use the `movhlps/movhlps` instructions to align the upper and lower parts of each register. This allows you to use a vertical add. With the resulting partial horizontal summation, full summation follows easily. [Figure 5-2](#) schematically presents horizontal add using `movhlps/movhlps`, while [Example 5-9](#) and [Example 5-10](#) provide the code for this operation.

Figure 5-2 Horizontal Add Using `movhlps/movhlps`



Example 5-9 Horizontal Add Using movhlps/movlhps

```

void horiz_add(Vertex_soa *in, float *out) {
    __asm {
        mov     ecx, in           // load structure addresses
        mov     edx, out
        movaps  xmm0, [ecx]       // load A1 A2 A3 A4 => xmm0
        movaps  xmm1, [ecx+16]    // load B1 B2 B3 B4 => xmm1
        movaps  xmm2, [ecx+32]    // load C1 C2 C3 C4 => xmm2
        movaps  xmm3, [ecx+48]    // load D1 D2 D3 D4 => xmm3

        // START HORIZONTAL ADD
        movaps  xmm5, xmm0        // xmm5= A1,A2,A3,A4
        movlhps xmm5, xmm1        // xmm5= A1,A2,B1,B2
        movhlps xmm1, xmm0        // xmm1= A3,A4,B3,B4
        addps   xmm5, xmm1        // xmm5= A1+A3,A2+A4,B1+B3,B2+B4
        movaps  xmm4, xmm2
        movlhps xmm2, xmm3        // xmm2= C1,C2,D1,D2
        movhlps xmm3, xmm4        // xmm3= C3,C4,D3,D4
        addps   xmm3, xmm2        // xmm3= C1+C3,C2+C4,D1+D3,D2+D4
        movaps  xmm6, xmm3        // xmm6= C1+C3,C2+C4,D1+D3,D2+D4
        shufps  xmm3, xmm5, 0xDD

                                //xmm6=A1+A3,B1+B3,C1+C3,D1+D3

        shufps  xmm5, xmm6, 0x88

                                // xmm5= A2+A4,B2+B4,C2+C4,D2+D4

        addps   xmm6, xmm5        // xmm6= D,C,B,A
        // END HORIZONTAL ADD
        movaps  [edx], xmm6
    }
}

```

Example 5-10 Horizontal Add Using Intrinsics with movhlps/movlhps

```

void horiz_add_intrin(Vertex_soa *in, float *out)
{
    __m128 v1, v2, v3, v4;
    __m128 tmm0, tmm1, tmm2, tmm3, tmm4, tmm5, tmm6;

    // Temporary variables
    tmm0 = _mm_load_ps(in->x); // tmm0 = A1 A2 A3 A4
    tmm1 = _mm_load_ps(in->y); // tmm1 = B1 B2 B3 B4
    tmm2 = _mm_load_ps(in->z); // tmm2 = C1 C2 C3 C4
    tmm3 = _mm_load_ps(in->w); // tmm3 = D1 D2 D3 D4
    tmm5 = tmm0; // tmm0 = A1 A2 A3 A4
    tmm5 = _mm_movelh_ps(tmm5, tmm1); // tmm5 = A1 A2 B1 B2
    tmm1 = _mm_movehl_ps(tmm1, tmm0); // tmm1 = A3 A4 B3 B4
    tmm5 = _mm_add_ps(tmm5, tmm1); // tmm5 = A1+A3 A2+A4 B1+B3 B2+B4
    tmm4 = tmm2;
    tmm2 = _mm_movelh_ps(tmm2, tmm3); // tmm2 = C1 C2 D1 D2
    tmm3 = _mm_movehl_ps(tmm3, tmm4); // tmm3 = C3 C4 D3 D4
    tmm3 = _mm_add_ps(tmm3, tmm2); // tmm3 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = tmm3; // tmm6 = C1+C3 C2+C4 D1+D3 D2+D4
    tmm6 = _mm_shuffle_ps(tmm3, tmm5, 0xDD);
    // tmm6 = A1+A3 B1+B3 C1+C3 D1+D3
    tmm5 = _mm_shuffle_ps(tmm5, tmm6, 0x88);
    // tmm5 = A2+A4 B2+B4 C2+C4 D2+D4
    tmm6 = _mm_add_ps(tmm6, tmm5);
    // tmm6 = A1+A2+A3+A4 B1+B2+B3+B4
    // C1+C2+C3+C4 D1+D2+D3+D4

    _mm_store_ps(out, tmm6);
}

```

Use of `cvttss2pi/cvttss2si` Instructions

The `cvttss2pi` and `cvttss2si` instructions encode the truncate/chop rounding mode implicitly in the instruction, thereby taking precedence over the rounding mode specified in the `MXCSR` register. This behavior can eliminate the need to change the rounding mode from round-nearest, to truncate/chop, and then back to round-nearest to resume computation. Frequent changes to the `MXCSR` register should be avoided since there is a penalty associated with writing this register; typically, through the use of the `cvttss2pi` and `cvttss2si` instructions, the rounding control in `MXCSR` can be always be set to round-nearest.

Flush-to-Zero Mode

Activating the flush-to-zero mode has the following effects during underflow situations:

- Precision and underflow exception flags are set to 1
- Zero result is returned

The IEEE mandated response to underflow is to deliver the denormalized result (that is, gradual underflow); consequently, the flush-to-zero mode is not compatible with IEEE Standard 754. It is provided to improve performance for applications where underflow is common and where the generation of a denormalized result is not necessary. Underflow for flush-to-zero mode occurs when the exponent for a computed result falls in the denormal range, regardless of whether a loss of accuracy has occurred.

Unmasking the underflow exception takes precedence over flush-to-zero mode. For a SSE instruction that generates an underflow condition an exception handler is invoked.

Optimizing Cache Usage for Intel Pentium 4 Processors

6

Over the past decade, processor speed has increased more than ten times, while memory access speed has increased only twice. This disparity makes it important to tune applications so that a majority of the data accesses are fulfilled in the processor caches. The performance of most applications can be considerably improved if the data they require can be fetched from the processor caches rather than from main memory.

Standard techniques to bring data into the processor before it is needed involves additional programming which can be difficult to implement and may require special steps to prevent performance degradation. The Streaming SIMD Extensions addressed these issues by providing the various prefetch instructions. The Intel Pentium 4 processor extends prefetching support via an automatic hardware data prefetch, a new mechanism for data prefetching based on current data access patterns that does not require programmer intervention.

Streaming SIMD Extensions also introduced the various non-temporal store instructions. Streaming SIMD Extensions 2 extend this support to the new data types, and also introduces non-temporal store support for the 32-bit integer registers.

This chapter focuses on two major subjects:

- Prefetch and Cacheability Instructions: discussion about the instructions that allow you to affect data caching in an application.
- Memory Optimization Using Prefetch and Cacheability Instructions: discussion and examples of various techniques for implementing memory optimizations using these instructions.



NOTE. *In a number of cases presented in this chapter, the prefetching and cache utilization are Pentium 4 processor platform-specific and may change for the future processors.*

General Prefetch Coding Guidelines

The following guidelines will help you optimize the usage of prefetches in your code (specific details will be discussed in subsequent sections):

- Use a current-generation compiler, such as the Intel C++ Compiler that supports C++ language-level features for the Streaming SIMD Extensions. The Streaming SIMD Extensions and MMX technology instructions provide intrinsics that allow you to optimize cache utilization. The examples of such Intel compiler intrinsics are `_mm_prefetch`, `_mm_stream` and `_mm_load`, `_mm_sfence`. For more details on these intrinsics, refer to the *Intel C++ Compiler User's Guide*, doc. number 718195.
- Facilitate compiler optimization:
 - Minimize use of global variables and pointers.
 - Minimize use of complex control flow.
 - Use the `const` modifier, avoid `register` modifier.
 - Choose data types carefully (see below) and avoid type casting.
- Optimize prefetch scheduling distance –
 - Far ahead enough to allow interim computation to overlap memory access time
 - Near enough that the prefetched data is not replaced from the data cache
- Use prefetch concatenation:
 - Arrange prefetches to avoid unnecessary prefetches at the end of an inner loop and to prefetch the first few iterations of the inner loop inside the next outer loop.

- Minimize the number of prefetches:
 - Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources. Excessive usage of prefetches can adversely impact application performance.
- Interleave prefetch with computation instructions:
 - For best performance, prefetch instructions must be interspersed with other computational instructions in the instruction sequence rather than clustered together.
- Use cache blocking techniques (for example, strip mining):
 - Improve cache hit rate by using cache blocking techniques such as strip-mining (one dimensional arrays) or loop blocking (two dimensional arrays)
- Balance single-pass versus multi-pass execution:
 - An algorithm can use single- or multi-pass execution defined as follows: single-pass, or unlayered execution passes a single data element through an entire computation pipeline. Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements before passing the entire batch on to the next stage.
 - General guideline: if your algorithm is single pass, use `prefetchnta`; if your algorithm is multi-pass use `prefetcht0`.
- Resolve memory bank conflict issues:
 - Minimize memory bank conflicts by applying array grouping to group contiguously used data together or allocating data within 4KB memory pages.
- Resolve cache management issues:
 - Minimize disturbance of temporal data held within the processor's caches by using streaming store instructions, as appropriate

Prefetch and Cacheability Instructions

The prefetch instruction, inserted by the programmers or compilers, accesses a minimum of one cache line of data (128 bytes on the Pentium 4 processor) prior to that data actually being needed. This hides the latency for data access in the time required

to process data already resident in the cache. Many algorithms can provide information in advance about the data that is to be required soon. In cases where the memory accesses are in long, regular data patterns, the automatic hardware prefetcher can hide memory access latency without the need for software prefetches.

The cacheability control instructions allow you to control data caching strategy in order to increase cache efficiency and minimize cache pollution.

Data reference patterns can be classified as follows:

Temporal	data will be used again soon
Spatial	data will be used in adjacent locations, for example, same cache line
Non-temporal	data which is referenced once and not reused in the immediate future; for example, some multimedia data types, such as the vertex buffer in a 3D graphics application.

These data characteristics are used in the discussions that follow.

Prefetch

This section discusses the mechanics of the software prefetch instructions and the automatic hardware prefetcher.

Software Data Prefetch

The `prefetch` instruction can hide the latency of data access in performance-critical sections of application code by allowing data to be fetched in advance of its actual usage. The `prefetch` instructions do not change the user-visible semantics of a program, although they may affect the program's performance. The `prefetch` instructions merely provide a hint to the hardware and generally will not generate exceptions or faults.

The `prefetch` instructions load either non-temporal data or temporal data in the specified cache level. This data access type and the cache level are specified as a hint. Depending on the implementation, the instruction fetches 32 or more aligned bytes, including the specified address byte, into the instruction-specified cache levels.

The `prefetch` instruction is implementation-specific; applications need to be tuned to each implementation to maximize performance.



NOTE. *Using the `prefetch` instructions is recommended only if data does not fit in cache.*

The `prefetch` instructions merely provide a hint to the hardware, and they will not generate exceptions or faults except for a few special cases (see the [“Prefetch and Load Instructions”](#) section). However, excessive use of `prefetch` instructions may waste memory bandwidth and result in performance penalty due to resource constraints.

Nevertheless, the `prefetch` instructions can lessen the overhead of memory transactions by preventing cache pollution and by using the caches and memory efficiently. This is particularly important for applications that share critical system resources, such as the memory bus. See an example in the [“Video Encoder”](#) section.

The `prefetch` instructions are mainly designed to improve application performance by hiding memory latency in the background. If segments of an application access data in a predictable manner, for example, using arrays with known strides, then they are good candidates for using `prefetch` to improve performance.

Use the `prefetch` instructions in:

- predictable memory access patterns
- time-consuming innermost loops
- locations where the execution pipeline may stall if data is not available.

Hardware Data Prefetch

The Pentium 4 processor implements an automatic data prefetcher which monitors application data access patterns and prefetches data automatically. This behavior is automatic and does not require programmer’s intervention.

Characteristics of the hardware data prefetcher are:

- Attempts to stay 256 bytes ahead of current data access locations

- Follows only one stream per 4K page (load or store)
- Can prefetch up to 8 simultaneous independent streams from eight different 4K regions
- Does not prefetch across 4K boundary; note that this is independent of paging modes.
- Fetches data into second/third-level cache
- Does not prefetch UC or WC memory types
- Follows load and store streams. Issues Read For Ownership (RFO) transactions for store streams and Data Reads for load streams.

The Prefetch Instructions – Pentium 4 Processor Implementation

Streaming SIMD Extensions include four flavors of `prefetch` instructions, one non-temporal, and three temporal. They correspond to two types of operations, temporal and non-temporal.



NOTE. *At the time of `prefetch`, if the data is already found in a cache level that is closer to the processor than the cache level specified by the instruction, no data movement occurs.*

The non-temporal instruction is

<code>prefetchnta</code>	Fetch the data into the second-level cache, minimizing cache pollution.
--------------------------	---

The temporal instructions are

<code>prefetcht0</code>	Fetch the data into all cache levels, that is, to the second-level cache for the Pentium 4 processor
<code>prefetcht1</code>	Identical to <code>prefetcht0</code>
<code>prefetcht2</code>	Identical to <code>prefetcht0</code>

[Table 6-1](#) lists the prefetch implementation differences between the Pentium III and Pentium 4 processors.

Table 6-1 Prefetch Implementation: Pentium III and Pentium 4 Processors

Prefetch Type	Pentium III processor	Pentium 4 processor
Prefetch NTA	Fetch 32 bytes	Fetch 128 bytes
	Fetch into 1st- level cache	Do not fetch into 1st-level cache
	Do not fetch into 2nd-level cache	Fetch into 1 way of 2nd-level cache
PrefetchT0	Fetch 32 bytes	Fetch 128 bytes
	Fetch into 1st- level cache	Do not fetch into 1st-level cache
	Fetch into 2nd- level cache	Fetch into 2nd- level cache
PrefetchT1, PrefetchT2	Fetch 32 bytes	Fetch 128 bytes
	Fetch into 2nd- level cache only	Do not fetch into 1st-level cache
	Do not fetch into 1st-level cache	Fetch into 2nd- level cache only

Prefetch and Load Instructions

The Pentium 4 processor has a decoupled execution and memory architecture that allows instructions to be executed independently with memory accesses if there are no data and resource dependencies. Programs or compilers can use dummy load instructions to imitate prefetch functionality, but preloading is not completely equivalent to prefetch instructions. Prefetch instructions provide a greater performance than preloading.

Currently, the `prefetch` instruction provides a greater performance gain than preloading because it:

- has no destination register, it only updates cache lines.
- does not stall the normal instruction retirement.
- does not affect the functional behavior of the program.
- has no cache line split accesses.
- does not cause exceptions except when `LOCK` prefix is used; the `LOCK` prefix is not a valid prefix for use with the `prefetch` instructions and should not be used.
- does not complete its own execution if that would cause a fault.

The current advantages of the prefetch over preloading instructions are processor-specific. The nature and extent of the advantages may change in the future.

In addition there are a few cases where a prefetch instruction will not perform the data prefetch if:

- the `prefetch` causes a DTLB (Data Translation Lookaside Buffer) miss.
- an access to the specified address causes a fault/exception.
- the memory subsystem runs out of request buffers between the first-level cache and the second-level cache.
- the `prefetch` targets an uncacheable memory region, for example, USWC and UC.
- a `LOCK` prefix is used. This causes an invalid opcode exception.

Cacheability Control

This section covers the mechanics of the cacheability control instructions.

The Non-temporal Store Instructions

This section describes the behavior of streaming stores and reiterates some of the information presented in the previous section. In Streaming SIMD Extensions, the `movntps`, `movntpd`, `movntq`, `movntdq`, `movnti`, `maskmovq` and `maskmovdqu` instructions are streaming, non-temporal stores. With regard to memory characteristics and ordering, they are similar mostly to the Write-Combining (wc) memory type:

- Write combining – successive writes to the same cache line are combined
- Write collapsing – successive writes to the same byte(s) result in only the last write being visible
- Weakly ordered – no ordering is preserved between wc stores, or between wc stores and other loads or stores
- Uncacheable and not write-allocating – stored data is written around the cache and will not generate a read-for-ownership bus request for the corresponding cache line.

Fencing

Because streaming stores are weakly ordered, a fencing operation is required to ensure that the stored data is flushed from the processor to memory. Failure to use an appropriate fence may result in data being “trapped” within the processor and will prevent visibility of this data by other processors or system agents. WC stores require software to ensure coherence of data by performing the fencing operation; see [“The fence Instructions”](#) section for more information.

Streaming Non-temporal Stores

Streaming stores can improve performance in the following ways:

- Increase store bandwidth since they do not require read-for-ownership bus requests
- Reduce disturbance of frequently used cached (temporal) data, since they write around the processor caches

Streaming stores allow cross-aliasing of memory types for a given memory region. For instance, a region may be mapped as write-back (WB) via the page attribute tables (PAT) or memory type range registers (MTRRS) and yet is written using a streaming store.

Memory Type and Non-temporal Stores

The memory type can take precedence over the non-temporal hint, leading to the following considerations:

- If the programmer specifies a non-temporal store to strongly-ordered uncacheable memory, for example, the Uncacheable (UC) or Write-Protect (WP) memory types, then the store behaves like an uncacheable store; the non-temporal hint is ignored and the memory type for the region is retained.
- If the programmer specifies the weakly-ordered uncacheable memory type of Write-Combining (WC), then the non-temporal store and the region have the same semantics, and there is no conflict.

- If the programmer specifies a non-temporal store to cacheable memory, for example, Write-Back (WB) or Write-Through (WT) memory types, two cases may result:
 1. If the data is present in the cache hierarchy, the instruction will ensure consistency. A particular processor may choose different ways to implement this. The following approaches are probable: (a) updating data in-place in the cache hierarchy while preserving the memory type semantics assigned to that region, or (b) evicting the data from the caches and writing the new non-temporal data to memory (with WC semantics). Pentium III processor implements a combination of both approaches.

If the streaming store hits a line that is present in the first-level cache, the store data will be combined in place within the first-level cache. If the streaming store hits a line present in the second-level, the line and stored data will be flushed from the second-level to system memory. Note that the approaches, separate or combined, can be different for future processors. Pentium 4 processor implements the latter policy, of evicting the data from all processor caches.

2. If the data is not present in the cache hierarchy, and the destination region is mapped as WB or WT, the transaction will be weakly ordered, and is subject to all WC memory semantics. The non-temporal store will not write-allocate. Different implementations may choose to collapse and combine these stores.

Write-Combining

Generally, WC semantics require software to ensure coherence, with respect to other processors and other system agents (such as graphics cards). Appropriate use of synchronization and a fencing operation (see [“The fence Instructions”](#) later in this chapter) must be performed for producer-consumer usage models. Fencing ensures that all system agents have global visibility of the stored data; for instance, failure to fence may result in a written cache line staying within a processor, and the line would not be visible to other agents.

For processors which implement non-temporal stores by updating data in-place that already resides in the cache hierarchy, the destination region should also be mapped as WC. Otherwise if mapped as WB or WT, there is a potential for speculative processor reads

to bring the data into the caches; in this case, non-temporal stores would then update in place, and data would not be flushed from the processor by a subsequent fencing operation.

The memory type visible on the bus in the presence of memory type aliasing is implementation-specific. As one possible example, the memory type written to the bus may reflect the memory type for the first store to this line, as seen in program order; other alternatives are possible. This behavior should be considered reserved, and dependence on the behavior of any particular implementation risks future incompatibility.

Streaming Store Usage Models

The two primary usage domains for streaming store are coherent requests and non-coherent requests.

Coherent Requests

Coherent requests are normal loads and stores to system memory, which may also hit cache lines present in another processor in a multi-processor environment. With coherent requests, a streaming store can be used in the same way as a regular store that has been mapped with a WC memory type (PAT or MTRR). An `sfence` instruction must be used within a producer-consumer usage model in order to ensure coherency and visibility of data between processors.

Within a single-processor system, the CPU can also re-read the same memory location and be assured of coherence (that is, a single, consistent view of this memory location): the same is true for a multi-processor (MP) system, assuming an accepted MP software producer-consumer synchronization policy is employed.

Non-coherent requests

Non-coherent requests arise from an I/O device, such as an AGP graphics card, that reads or writes system memory using non-coherent requests, which are not reflected on the processor bus and thus will not query the processor's caches. An `sfence` instruction must be used within a producer-consumer usage model in order to ensure coherency and visibility of data between processors. In this case, if the processor is

writing data to the I/O device, a streaming store can be used with a processor with any behavior of approach (a), [page 6-10](#), above, only if the region has also been mapped with a WC memory type (PAT, MTRR).



CAUTION. *Failure to map the region as WC may allow the line to be speculatively read into the processor caches, that is, via the wrong path of a mispredicted branch.*

In case the region is not mapped as WC, the streaming might update in-place in the cache and a subsequent `sfence` would not result in the data being written to system memory. Explicitly mapping the region as WC in this case ensures that any data read from this region will not be placed in the processor's caches. A read of this memory location by a non-coherent I/O device would return incorrect/out-of-date results. For a processor which solely implements approach (b), [page 6-10](#), above, a streaming store can be used in this non-coherent domain without requiring the memory region to also be mapped as WB, since any cached data will be flushed to memory by the streaming store.

Streaming Store Instruction Descriptions

The `movntq/movntdq` (non-temporal store of packed integer in an MMX technology or Streaming SIMD Extensions register) instructions store data from a register to memory. The instruction is implicitly weakly-ordered, does no write-allocate, and so minimizes cache pollution.

The `movntps` (non-temporal store of packed single precision floating point) instruction is similar to `movntq`. It stores data from a Streaming SIMD Extensions register to memory in 16-byte granularity. Unlike `movntq`, the memory address must be aligned to a 16-byte boundary or a general protection exception will occur. The instruction is implicitly weakly-ordered, does not write-allocate, and thus minimizes cache pollution.

The `maskmovq/maskmovdqu` (non-temporal byte mask store of packed integer in an MMX technology or Streaming SIMD Extensions register) instructions store data from a register to the location specified by the `edi` register. The most significant bit in each

byte of the second mask register is used to selectively write the data of the first register on a per-byte basis. The instruction is implicitly weakly-ordered (that is, successive stores may not write memory in original program-order), does not write-allocate, and thus minimizes cache pollution.

The fence Instructions

The following fence instructions are available: `sfence`, `lfence`, and `mfence`.

The `sfence` Instruction

The `sfence` (store fence) instruction makes it possible for every `store` instruction that precedes the `sfence` instruction in program order to be globally visible before any `store` instruction that follows the `sfence`. The `sfence` instruction provides an efficient way of ensuring ordering between routines that produce weakly-ordered results.

The use of weakly-ordered memory types can be important under certain data sharing relationships, such as a producer-consumer relationship. Using weakly-ordered memory can make assembling the data more efficient, but care must be taken to ensure that the consumer obtains the data that the producer intended to see. Some common usage models may be affected in this way by weakly-ordered stores. Examples are:

- library functions, which use weakly-ordered memory to write results
- compiler-generated code, which also benefits from writing weakly-ordered results
- hand-crafted code

The degree to which a consumer of data knows that the data is weakly-ordered can vary for these cases. As a result, the `sfence` instruction should be used to ensure ordering between routines that produce weakly-ordered data and routines that consume this data. The `sfence` instruction provides a performance-efficient way by ensuring the ordering when every `store` instruction that precedes the `store fence` instruction in program order is globally visible before any `store` instruction which follows the `fence`.

The `lfence` Instruction

The `lfence` (load fence) instruction makes it possible for every load instruction that precedes the `lfence` instruction in program order to be globally visible before any load instruction that follows the `lfence`. The `lfence` instruction provides a means of segregating certain load instructions from other loads.

The `mfence` Instruction

The `mfence` (memory fence) instruction makes it possible for every load and store instruction that precedes the `mfence` instruction in program order to be globally visible before any other load or store instruction that follows the `mfence`. The `mfence` instruction provides a means of segregating certain memory instructions from other memory references.

Note that the use of a `lfence` and `sfence` is not equivalent to the use of a `mfence` since the load and store fences are not ordered with respect to each other. In other words, the load fence can be executed before prior stores, and the store fence can be executed before prior loads. The `mfence` instruction should be used whenever the cache line flush instruction (`clflush`) is used to ensure that speculative memory references generated by the processor do not interfere with the flush; see [“The `clflush` Instruction”](#) for more information.

The `clflush` Instruction

The cache line associated with the linear address specified by the value of byte address is invalidated from all levels of the processor cache hierarchy (data and instruction). The invalidation is broadcast throughout the coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty) it is written to memory before invalidation. Other characteristics include:

- The data size affected is the cache coherency size, which is 64 bytes on Pentium 4 processor.
- The memory attribute of the page containing the affected line has no effect on the behavior of this instruction.
- The `clflush` instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load.

`clflush` is an unordered operation with respect to other memory traffic including other `clflush` instructions. Software should use a `mfence`, memory fence for cases where ordering is a concern.

As an example, consider a video usage model, wherein a video capture device is using non-coherent AGP accesses to write a capture stream directly to system memory. Since these non-coherent writes are not broadcast on the processor bus, they will not flush any copies of the same locations that reside in the processor caches. As a result, before the processor re-reads the capture buffer, it should use `clflush` to ensure that any stale copies of the capture buffer are flushed from the processor caches. Due to speculative reads that may be generated by the processor, it is important to observe appropriate fencing, using `mfence`. [Example 6-1](#) illustrates the pseudo-code for the recommended usage of `clflush`:

Example 6-1 Pseudo-code for Using `clflush`

```
while (!buffer_ready) {}  
mfence  
    for(i=0;i<num_cachelines;i+=cacheline_size) {  
        clflush (char *)((unsigned int)buffer + i)  
    }  
mfence  
    prefnta buffer[0];  
    VAR = buffer[0];
```

Memory Optimization Using Prefetch

The Pentium 4 processor has two mechanisms for data prefetch: software-controlled prefetch and an automatic hardware prefetch.

Software-controlled Prefetch

The software-controlled prefetch is enabled using the four prefetch instructions introduced with Streaming SIMD Extensions instructions. These instructions are hints to bring a cache line of data in to various levels and modes in the cache hierarchy. The software-controlled prefetch is not intended for prefetching code. Using it can incur significant penalties on a multiprocessor system when code is shared.

Software prefetching has the following characteristics:

- Can handle irregular access patterns, which do not trigger the hardware prefetcher.
- Can use less bus bandwidth than hardware prefetching; see below.
- Software prefetches must be added to new code, and do not benefit existing applications.

Hardware Prefetch

The automatic hardware prefetch, can bring lines into the unified first-level cache based on prior data misses. The automatic hardware prefetcher will attempt to prefetch two cache lines ahead of the prefetch stream. This feature is introduced with the Pentium 4 processor.

There are different strengths and weaknesses to software and hardware prefetching of the Pentium 4 processor. The characteristics of the hardware prefetching are as follows (compare with the software prefetching features listed above):

- Works with existing applications.
- Requires regular access patterns.
- Start-up penalty before hardware prefetcher triggers and extra fetches after array finishes. For short arrays this overhead can reduce effectiveness of the hardware prefetcher.

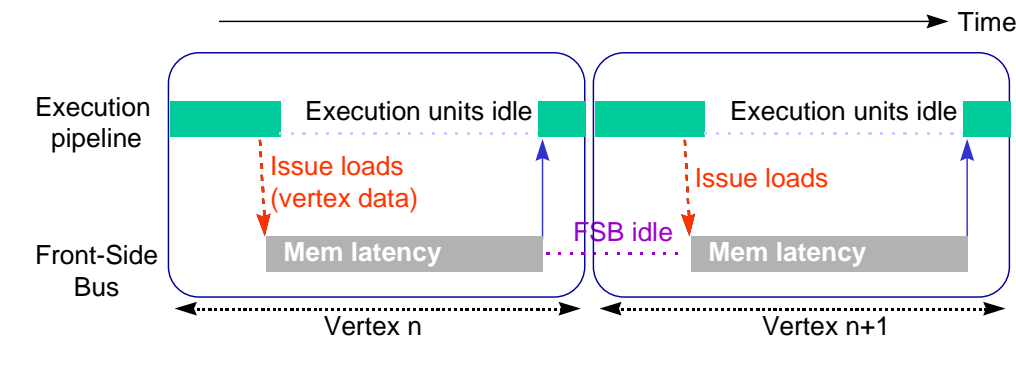
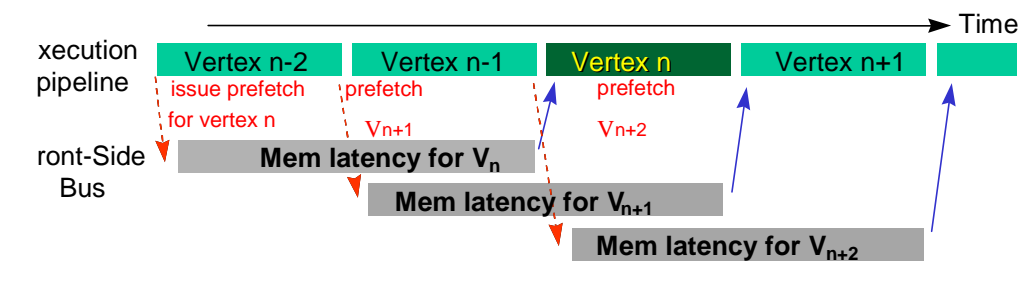
- The hardware prefetcher requires a couple misses before it starts operating.
- Hardware prefetching will generate a request for data beyond the end of an array, which will not be utilized. This behavior wastes bus bandwidth. In addition this behavior results in a start-up penalty when fetching the beginning of the next array; this occurs because the wasted prefetch should have been used instead to hide the latency for the initial data in the next array. Software prefetching can recognize and handle these cases.
- Will not prefetch across a 4K page boundary i.e. the program would have to initiate demand loads for the new page before the hardware prefetcher will start prefetching from the new page.

Example of Latency Hiding with S/W Prefetch Instruction

Achieving the highest level of memory optimization using prefetch instructions requires an understanding of the micro-architecture and system architecture of a given machine. This section translates the key architectural implications into several simple guidelines for programmers to use.

[Figure 6-1](#) and [Figure 6-2](#) show two scenarios of a simplified 3D geometry pipeline as an example. A 3D-geometry pipeline typically fetches one vertex record at a time and then performs transformation and lighting functions on it. Both figures show two separate pipelines, an execution pipeline, and a memory pipeline (front-side bus).

Since the Pentium 4 processor, similarly to the Pentium II and Pentium III processors, completely decouples the functionality of execution and memory access, these two pipelines can function concurrently. [Figure 6-1](#) shows “bubbles” in both the execution and memory pipelines. When loads are issued for accessing vertex data, the execution units sit idle and wait until data is returned. On the other hand, the memory bus sits idle while the execution units are processing vertices. This scenario severely decreases the advantage of having a decoupled architecture.

Figure 6-1 Memory Access Latency and Execution Without Prefetch**Figure 6-2 Memory Access Latency and Execution With Prefetch**

The performance loss caused by poor utilization of resources can be completely eliminated by correctly scheduling the prefetch instructions appropriately. As shown in [Figure 6-2](#), prefetch instructions are issued two vertex iterations ahead. This assumes that only one vertex gets processed in one iteration and a new data cache line is needed for each iteration. As a result, when iteration n , vertex V_n , is being processed, the requested data is already brought into cache. In the meantime, the front-side bus is transferring the data needed for iteration $n+1$, vertex V_{n+1} . Because there is no dependence between V_{n+1} data and the execution of V_n , the latency for data access of

V_{n+1} can be entirely hidden behind the execution of V_n . Under such circumstances, no “bubbles” are present in the pipelines and thus the best possible performance can be achieved.

Prefetching is useful for inner loops that have heavy computations, or are close to the boundary between being compute-bound and memory-bandwidth-bound.

The prefetch is probably not very useful for loops which are predominately memory bandwidth-bound.

When data is already located in the first level cache, prefetching can be useless and could even slow down the performance because the extra μ ops either back up waiting for outstanding memory accesses or may be dropped altogether. This behavior is platform-specific and may change in the future.

Prefetching Usage Checklist

The following checklist covers issues that need to be addressed and/or resolved to use the prefetch instruction properly:

- Determine prefetch scheduling distance
- Use prefetch concatenation
- Minimize the number of prefetches
- Mix prefetch with computation instructions
- Use cache blocking techniques (for example, strip mining)
- Balance single-pass versus multi-pass execution
- Resolve memory bank conflict issues
- Resolve cache management issues

The subsequent sections discuss all the above items.

Prefetch Scheduling Distance

Determining the ideal prefetch placement in the code depends on many architectural parameters, including the amount of memory to be prefetched, cache lookup latency, system memory latency, and estimate of computation cycle. The ideal distance for prefetching data is processor- and platform- dependent. If the distance is too short, the

prefetch will not hide any portion of the latency of the fetch behind computation. If the prefetch is too far ahead, the prefetched data may be flushed out of the cache by the time it is actually required.

Since prefetch distance is not a well-defined metric, for this discussion, we define a new term, prefetch scheduling distance (PSD), which is represented by the number of iterations. For large loops, prefetch scheduling distance can be set to 1, that is, schedule prefetch instructions one iteration ahead. For small loop bodies, that is, loop iterations with little computation, the prefetch scheduling distance must be more than one iteration.

A simplified equation to compute PSD is deduced from the mathematical model. For a simplified equation, complete mathematical model, and methodology of prefetch distance determination, refer to [Appendix E, “Mathematics of Prefetch Scheduling Distance”](#).

[Example 6-2](#) illustrates the use of a prefetch within the loop body. The prefetch scheduling distance is set to 3, `esi` is effectively the pointer to a line, `edx` is the address of the data being referenced and `xmm1-xmm4` are the data used in computation.

[Example 6-2](#) uses two independent cache lines of data per iteration. The PSD would need to be increased/decreased if more/less than two cache lines are used per iteration.

Example 6-2 Prefetch Scheduling Distance

```
top_loop:
    prefetchnta [edx + esi + 128*3]
    prefetchnta [edx*4 + esi + 128*3]
    . . . . .
    movaps xmm1, [edx + esi]
    movaps xmm2, [edx*4 + esi]
    movaps xmm3, [edx + esi + 16]
    movaps xmm4, [edx*4 + esi + 16]
    . . . . .
    . . . . .
    add     esi, 128
    cmp     esi, ecx
    jl      top_loop
```

Prefetch Concatenation

Maximum performance can be achieved when execution pipeline is at maximum throughput, without incurring any memory latency penalties. This can be achieved by prefetching data to be used in successive iterations in a loop. De-pipelining memory generates bubbles in the execution pipeline. To explain this performance issue, a 3D geometry pipeline that processes 3D vertices in strip format is used as an example. A strip contains a list of vertices whose predefined vertex order forms contiguous triangles. It can be easily observed that the memory pipe is de-pipelined on the strip boundary due to ineffective prefetch arrangement. The execution pipeline is stalled for the first two iterations for each strip. As a result, the average latency for completing an iteration will be 165(FIX) clocks. (See [Appendix E, “Mathematics of Prefetch Scheduling Distance”](#), for a detailed memory pipeline description.)

This memory de-pipelining creates inefficiency in both the memory pipeline and execution pipeline. This de-pipelining effect can be removed by applying a technique called prefetch concatenation. With this technique, the memory access and execution can be fully pipelined and fully utilized.

For nested loops, memory de-pipelining could occur during the interval between the last iteration of an inner loop and the next iteration of its associated outer loop. Without paying special attention to prefetch insertion, the loads from the first iteration of an inner loop can miss the cache and stall the execution pipeline waiting for data returned, thus degrading the performance.

In the code of [Example 6-3](#), the cache line containing `a[ii][0]` is not prefetched at all and always misses the cache. This assumes that no array `a[][]` footprint resides in the cache. The penalty of memory de-pipelining stalls can be amortized across the inner loop iterations. However, it may become very harmful when the inner loop is short. In addition, the last prefetch in the last PSD iterations are wasted and consume machine resources. Prefetch concatenation is introduced here in order to eliminate the performance issue of memory de-pipelining.

Example 6-3 Using Prefetch Concatenation

```
for (ii = 0; ii < 100; ii++) {  
    for (jj = 0; jj < 32; jj+=8) {  
        prefetch a[ii][jj+8]  
        computation a[ii][jj]  
    }  
}
```

Prefetch concatenation can bridge the execution pipeline bubbles between the boundary of an inner loop and its associated outer loop. Simply by unrolling the last iteration out of the inner loop and specifying the effective prefetch address for data used in the following iteration, the performance loss of memory de-pipelining can be completely removed. [Example 6-4](#) gives the rewritten code.

Example 6-4 Concatenation and Unrolling the Last Iteration of Inner Loop

```
for (ii = 0; ii < 100; ii++) {  
    for (jj = 0; jj < 24; jj+=8) { /* N-1 iterations */  
        prefetch a[ii][jj+8]  
        computation a[ii][jj]  
    }  
    prefetch a[ii+1][0]  
    computation a[ii][jj]/* Last iteration */  
}
```

This code segment for data prefetching is improved and only the first iteration of the outer loop suffers any memory access latency penalty, assuming the computation time is larger than the memory latency. Inserting a prefetch of the first data element needed prior to entering the nested loop computation would eliminate or reduce the start-up penalty for the very first iteration of the outer loop. This uncomplicated high-level code optimization can improve memory performance significantly.

Minimize Number of Prefetches

Prefetch instructions are not completely free in terms of bus cycles, machine cycles and resources, even though they require minimal clocks and memory bandwidth.

Excessive prefetching may lead to performance penalties because of issue penalties in the front-end of the machine and/or resource contention in the memory sub-system. This effect may be severe in cases where the target loops are small and/or cases where the target loop is issue-bound.

One approach to solve the excessive prefetching issue is to unroll and/or software-pipeline the loops to reduce the number of prefetches required. [Figure 6-3](#) presents a code example which implements prefetch and unrolls the loop to remove the redundant prefetch instructions whose prefetch addresses hit the previously issued prefetch instructions. In this particular example, unrolling the original loop once saves six prefetch instructions and nine instructions for conditional jumps in every other iteration.

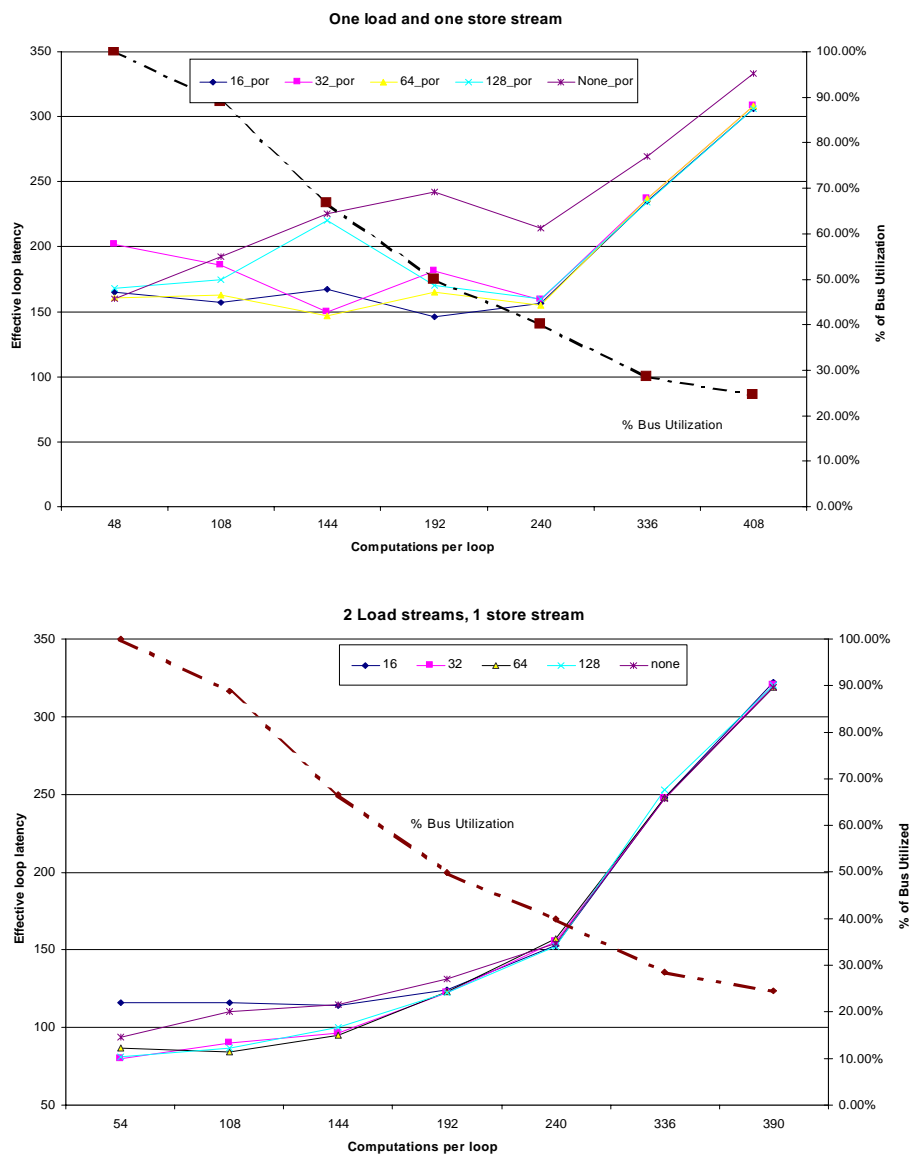
Figure 6-3 Prefetch and Loop Unrolling

<pre> top_loop: prefetchnta [edx+esi+32] prefetchnta [edx*4+esi+32] movaps xmm1, [edx+esi] movaps xmm2, [edx*4+esi] add esi, 16 cmp esi, ecx jl top_loop </pre>	<pre> top_loop: prefetchnta [edx+esi+128] prefetchnta [edx*4+esi+128] movaps xmm1, [edx+esi] movaps xmm2, [edx*4+esi] movaps xmm1, [edx+esi+16] movaps xmm2, [edx*4+esi+16] movaps xmm1, [edx+esi+96] movaps xmm2, [edx*4+esi+96] add esi, 128 cmp esi, ecx jl top_loop </pre>
---	--

[Figure 6-4](#) demonstrates the effectiveness of software prefetches in latency hiding. The X axis indicates the number of computation clocks per loop (each iteration is independent). The Y axis indicates the execution time measured in clocks per loop. The secondary Y axis indicates the percentage of bus bandwidth utilization. The tests vary by the following parameters:

1. The number of load/store streams. Each load and store stream accesses one 128-byte cache line each, per iteration.
2. The amount of computation per loop. This is varied by increasing the number of dependent arithmetic operations executed.
3. The number of the software prefetches per loop. (for example, one every 16 bytes, 32 bytes, 64 bytes, 128 bytes).

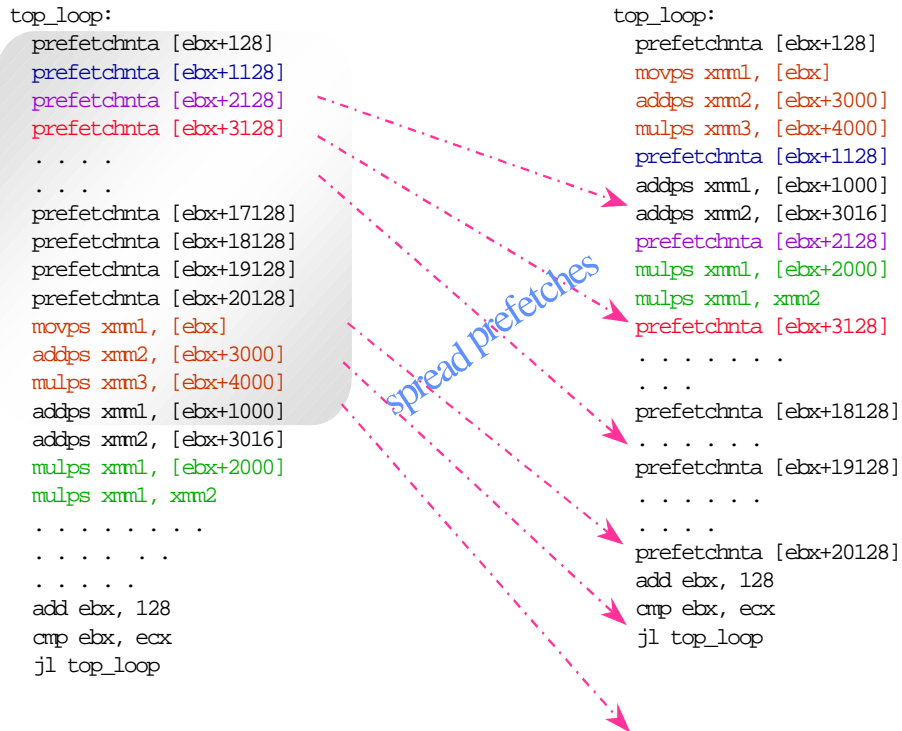
As expected, the leftmost portion of each of the graphs in shows that when there is not enough computation to overlap the latency of memory access, prefetch does not help and that the execution is essentially memory- bound. The graphs also illustrate that redundant prefetches do not increase performance.

Figure 6-4 Memory Access Latency and Execution With Prefetch

Mix Prefetch with Computation Instructions

It may seem convenient to cluster all of the prefetch instructions at the beginning of a loop body or before a loop, but this can lead to severe performance degradation. In order to achieve best possible performance, prefetch instructions must be interspersed with other computational instructions in the instruction sequence rather than clustered together. If possible, they should also be placed apart from loads. This improves the instruction level parallelism and reduces the potential instruction resource stalls. In addition, this mixing reduces the pressure on the memory access resources and in turn reduces the possibility of the prefetch retiring without fetching data.

[Example 6-5](#) illustrates distributing prefetch instructions. A simple and useful heuristic of prefetch spreading for a Pentium 4 processor is to insert a prefetch instruction every 20 to 25 clocks. Rearranging prefetch instructions could yield a noticeable speedup for the code which stresses the cache resource.

Example 6-5 Spread Prefetch Instructions

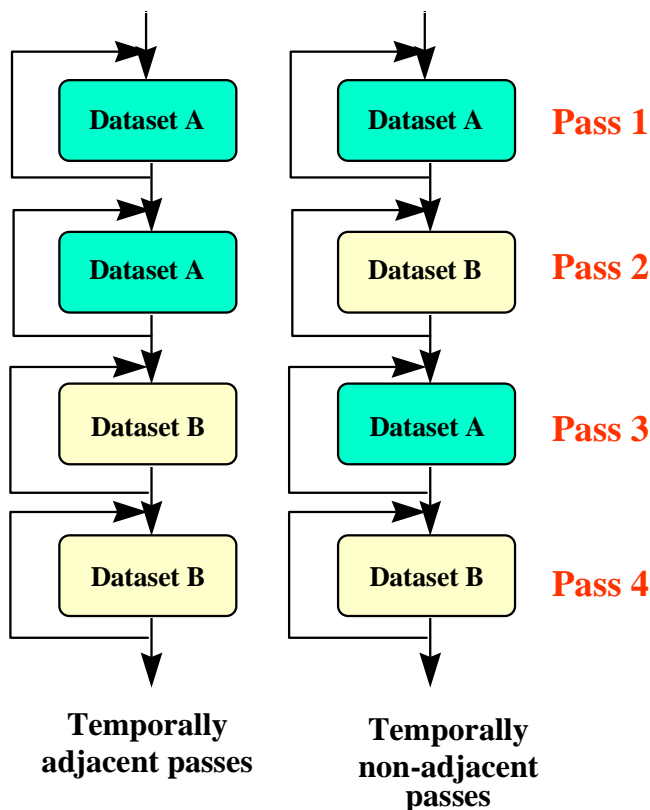
NOTE. To avoid instruction execution stalls due to the over-utilization of the resource, prefetch instructions must be interspersed with computational instructions.

Prefetch and Cache Blocking Techniques

Cache blocking techniques, such as strip-mining, are used to improve temporal locality, and thereby cache hit rate. Strip-mining is a one-dimensional temporal locality optimization for memory. When two-dimensional arrays are used in programs, loop blocking technique (similar to strip-mining but in two dimensions) can be applied for a better memory performance.

If an application uses a large data set that can be reused across multiple passes of a loop, it will benefit from strip mining: data sets larger than the cache will be processed in groups small enough to fit into cache. This allows temporal data to reside in the cache longer, reducing bus traffic.

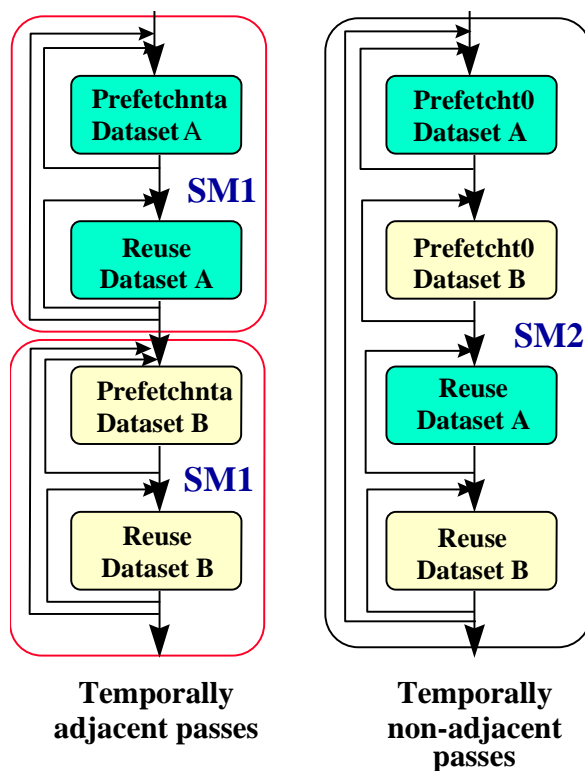
Data set size and temporal locality (data characteristics) fundamentally affect how prefetch instructions are applied to strip-mined code. [Figure 6-5](#) shows two simplified scenarios for temporally-adjacent data and temporally-non-adjacent data.

Figure 6-5 Cache Blocking – Temporally Adjacent and Non-adjacent Passes

In the temporally-adjacent scenario, subsequent passes use the same data and find it already in second-level cache. Prefetch issues aside, this is the preferred situation. In the temporally non-adjacent scenario, data used in pass m is displaced by pass $(m+1)$, requiring data *re-fetch* into the first level cache and perhaps the second level cache if a later pass reuses the data. If both data sets fit into the second-level cache, load operations in passes 3 and 4 become less expensive.

[Figure 6-6](#) shows how prefetch instructions and strip-mining can be applied to increase performance in both of these scenarios.

Figure 6-6 Examples of Prefetch and Strip-mining for Temporally Adjacent and Non-Adjacent Passes Loops



For Pentium 4 processors, the left scenario shows a graphical implementation of using `prefetchnta` to prefetch data into selected ways of the second-level cache *only* (SM1 denotes strip mine one way of second-level), minimizing second-level cache pollution. Use `prefetchnta` if the data is only touched once during the entire execution pass in

order to minimize cache pollution in the higher level caches. This provides instant availability, assuming the prefetch was issued far ahead enough, when the read access is issued.

In scenario to the right, in [Figure 6-6](#), keeping the data in one way of the second-level cache does not improve cache locality. Therefore, use `prefetcht0` to prefetch the data. This hides the latency of the memory references in passes 1 and 2, and keeps a copy of the data in second-level cache, which reduces memory traffic and latencies for passes 3 and 4. To further reduce the latency, it might be worth considering extra `prefetchnta` instructions prior to the memory references in passes 3 and 4.

In [Example 6-6](#), consider the data access patterns of a 3D geometry engine first without strip-mining and then incorporating strip-mining. Note that 4-wide SIMD instructions of Pentium III processor can process 4 vertices per every iteration.

Example 6-6 Data Access of a 3D Geometry Engine without Strip-mining

```
while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data      // v =[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    TRANSFORMATION code           // use only x,y,z,tu,tv of a vertex
    nvtx+=4
}

while (nvtx < MAX_NUM_VTX) {
    prefetchnta vertexi data      // v =[x,y,z,nx,ny,nz,tu,tv]
                                   // x,y,z fetched again

    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    compute the light vectors     // use only x,y,z
    LOCAL LIGHTING code           // use only nx,ny,nz
    nvtx+=4
}
```

Without strip-mining, all the x,y,z coordinates for the four vertices must be re-fetched from memory in the second pass, that is, the lighting loop. This causes under-utilization of cache lines fetched during transformation loop as well as bandwidth wasted in the lighting loop.

Now consider the code in [Example 6-7](#) where strip-mining has been incorporated into the loops.

Example 6-7 Data Access of a 3D Geometry Engine with Strip-mining

```
while (nstrip < NUM_STRIP) {
/* Strip-mine the loop to fit data into one way of the second-level
   cache */
while (nvtx < MAX_NUM_VTX_PER_STRIP) {
    prefetchnta vertexi data          // v=[x,y,z,nx,ny,nz,tu,tv]
    prefetchnta vertexi+1 data
    prefetchnta vertexi+2 data
    prefetchnta vertexi+3 data
    TRANSFORMATION code
    nvtx+=4
}
while (nvtx < MAX_NUM_VTX_PER_STRIP) {
    /* x y z coordinates are in the second-level cache, no prefetch is
       required */
    compute the light vectors
    POINT LIGHTING code
    nvtx+=4
}
}
```

With strip-mining, all the vertex data can be kept in the cache (for example, one way of second-level cache) during the strip-mined transformation loop and reused in the lighting loop. Keeping data in the cache reduces both bus traffic and the number of prefetches used.

[Figure 6-7](#) summarizes the steps of the basic usage model that incorporates prefetch with strip-mining. The steps are:

- Do strip-mining: partition loops so that the dataset fits into second-level cache.
- Use `prefetchnta` if the data is only used once or the dataset fits into 32K (one way of second-level cache). Use `prefetcht0` if the dataset exceeds 32K.

The above steps are platform-specific and provide an implementation example. The variables `NUM_STRIP` and `MAX_NUM_VX_PER_STRIP` can be heuristically determined for peak performance for specific application on a specific platform.

Figure 6-7 Incorporating Prefetch into Strip-mining Code

Use Once	Use Multiple Times	
	Adjacent Passes	Non-Adjacent Passes
Prefetchnta	Prefetch0, SM1	Prefetch0, SM1 (2 nd Level Pollution)

Single-pass versus Multi-pass Execution

An algorithm can use single- or multi-pass execution defined as follows:

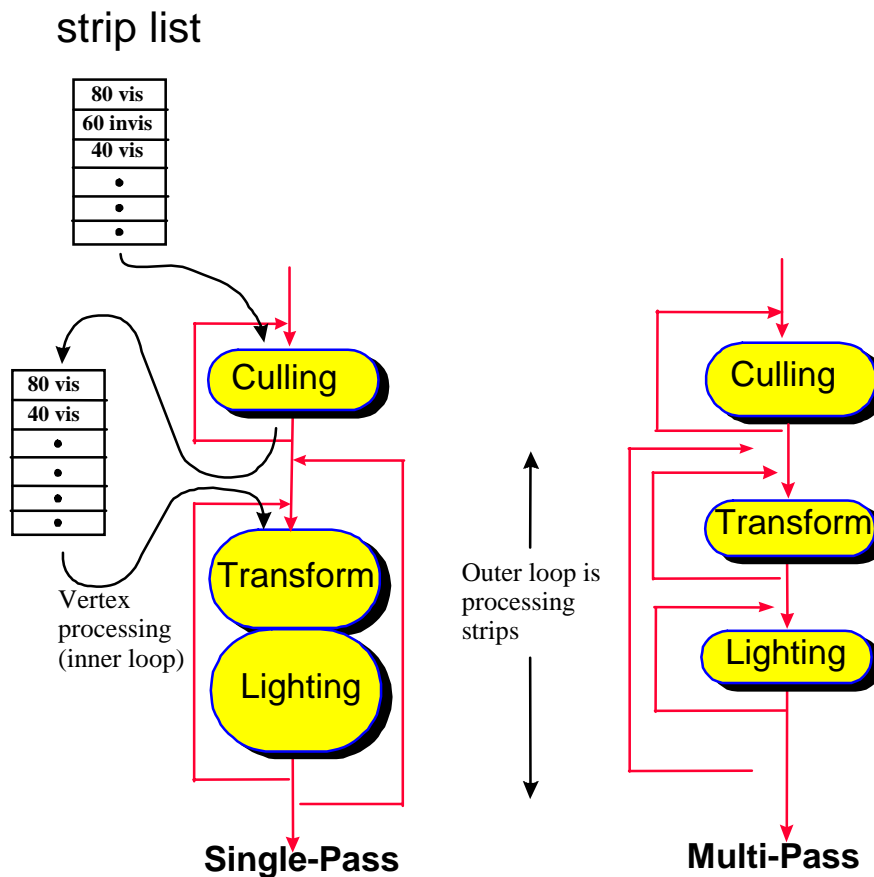
- Single-pass, or unlayered execution passes a single data element through an entire computation pipeline.
- Multi-pass, or layered execution performs a single stage of the pipeline on a batch of data elements, before passing the batch on to the next stage.

A characteristic feature of both single-pass and multi-pass execution is that a specific trade-off exists depending on an algorithm's implementation and use of a single-pass or multiple-pass execution, see [Figure 6-8](#).

Multi-pass execution is often easier to use when implementing a general purpose API, where the choice of code paths that can be taken depends on the specific combination of features selected by the application (for example, for 3D graphics, this might include the type of vertex primitives used and the number and type of light sources).

With such a broad range of permutations possible, a single-pass approach would be complicated, in terms of code size and validation. In such cases, each possible permutation would require a separate code sequence. For example, an object with features A, B, C, D can have a subset of features enabled, say, A, B, D. This stage would use one code path; another combination of enabled features would have a different code path. It makes more sense to perform each pipeline stage as a separate pass, with conditional clauses to select different features that are implemented within each stage. By using strip-mining, the number of vertices processed by each stage (for example, the batch size) can be selected to ensure that the batch stays within the processor caches through all passes. An intermediate cached buffer is used to pass the batch of vertices from one stage or pass to the next one.

Single-pass execution can be better suited to applications which limit the number of features that may be used at a given time. A single-pass approach can reduce the amount of data copying that can occur with a multi-pass engine, see [Figure 6-8](#).

Figure 6-8 Single-Pass Vs. Multi-Pass 3D Geometry Engines

The choice of single-pass or multi-pass can have a number of performance implications. For instance, in a multi-pass pipeline, stages that are limited by bandwidth (either input or output) will reflect more of this performance limitation in overall execution time. In contrast, for a single-pass approach, bandwidth-limitations

can be distributed/amortized across other computation-intensive stages. Also, the choice of which prefetch hints to use are also impacted by whether a single-pass or multi-pass approach is used (see [“Prefetch and Cacheability Instructions”](#)).

Memory Optimization using Non-Temporal Stores

The non-temporal stores can also be used to manage data retention in the cache. Uses for the non-temporal stores include:

- To combine many writes without disturbing the cache hierarchy
- To manage which data structures remain in the cache and which are transient.

Detailed implementations of these usage models are covered in the following sections.

Non-temporal Stores and Software Write-Combining

Use non-temporal stores in the cases when the data to be stored is:

- write-once (non-temporal)
- too large and thus cause cache thrashing.

Non-temporal stores do not invoke a cache line allocation, which means they are not write-allocate. As a result, caches are not polluted and no dirty writeback is generated to compete with useful data bandwidth. Without using non-temporal stores, bus bandwidth will suffer when caches start to be thrashed because of dirty writebacks.

In Streaming SIMD Extensions implementation, when non-temporal stores are written into writeback or write-combining memory regions, these stores are weakly-ordered and will be combined internally inside the processor’s write-combining buffer and be written out to memory as a line burst transaction. To achieve the best possible performance, it is recommended to align data along the cache line boundary and write them consecutively in a cache line size while using non-temporal stores. If the consecutive writes are prohibitive due to programming constraints, then software write-combining (SWWC) buffers can be used to enable line burst transaction.

You can declare small SWWC buffers (a cache line for each buffer) in your application to enable explicit write-combining operations. Instead of writing to non-temporal memory space immediately, the program writes data into SWWC buffers and combines them inside these buffers. The program only writes a SWWC buffer out using

non-temporal stores when the buffer is filled up, that is, a cache line (128 bytes for the Pentium 4 processor). Although the `SWWC` method requires explicit instructions for performing temporary writes and reads, this ensures that the transaction on the front-side bus causes line transaction rather than several partial transactions. Application performance gains considerably from implementing this technique. These `SWWC` buffers can be maintained in the second-level and re-used throughout the program.

Cache Management

The streaming instructions (`prefetch` and `stores`) can be used to manage data and minimize disturbance of temporal data held within the processor's caches.

In addition, the Pentium 4 processor takes advantage of the Intel C++ Compiler that supports C++ language-level features for the Streaming SIMD Extensions. The Streaming SIMD Extensions and MMX technology instructions provide intrinsics that allow you to optimize cache utilization. The examples of such Intel compiler intrinsics are `_mm_prefetch`, `_mm_stream`, `_mm_load`, `_mm_sfence`. For more details on these intrinsics, refer to the *Intel C++ Compiler User's Guide*, order number 718195.

The following examples of using prefetching instructions in the operation of video encoder and decoder as well as in simple 8-byte memory copy, illustrate performance gain from using the prefetching instructions for efficient cache management.

Video Encoder

In a video encoder example, some of the data used during the encoding process is kept in the processor's second-level cache, to minimize the number of reference streams that must be re-read from system memory. To ensure that other writes do not disturb the data in the second-level cache, streaming stores (`movntq`) are used to write around all processor caches.

The prefetching cache management implemented for the video encoder reduces the memory traffic. The second-level cache pollution reduction is ensured by preventing single-use video frame data from entering the second-level cache. Using a non-temporal prefetch (`prefetchnta`) instruction brings data into only one way of the second-level cache, thus reducing pollution of the second-level cache. If the data brought directly to second-level cache is not re-used, then there is a performance gain

from the non-temporal prefetch over a temporal prefetch. The encoder uses non-temporal prefetches to avoid pollution of the second-level cache, increasing the number of second-level cache hits and decreasing the number of polluting write-backs to memory. The performance gain results from the more efficient use of the second-level cache, not only from the prefetch itself.

Video Decoder

In the video decoder example, completed frame data is written to local memory of the graphics card, which is mapped to `WC` (Write-combining) memory type. A copy of reference data is stored to the `WB` memory at a later time by the processor in order to generate future data. The assumption is that the size of the reference data is too large to fit in the processor's caches. A streaming store is used to write the data around the cache, to avoid displaying other temporal data held in the caches. Later, the processor re-reads the data using `prefetchnta`, which ensures maximum bandwidth, yet minimizes disturbance of other cached temporal data by using the non-temporal (NTA) version of prefetch.

Conclusions from Video Encoder and Decoder Implementation

These two examples indicate that by using an appropriate combination of non-temporal prefetches and non-temporal stores, an application can be designed to lessen the overhead of memory transactions by preventing second-level cache pollution, keeping useful data in the second-level cache and reducing costly write-back transactions. Even if an application does not gain performance significantly from having data ready from prefetches, it can improve from more efficient use of the second-level cache and memory. Such design reduces the encoder's demand for such critical resource as the memory bus. This makes the system more balanced, resulting in higher performance.

Using Prefetch and Streaming-store for a Simple Memory Copy

Consider a memory copy task to transfer a large array of 8-byte data elements from one memory location to another. [Example 6-8](#) presents the basic algorithm of the simple memory copy. This task can be sped up greatly using prefetch and streaming store instructions. The techniques are discussed in the following paragraph and a code example is shown in [Example 6-9](#).

Example 6-8 Basic Algorithm of a Simple Memory Copy

```
#define N 512000
double a[N], b[N];
for (i = 0; i < N; i++) {
    b[i] = a[i];
}
```

The memory copy algorithm can be optimized using the Streaming SIMD Extensions and these considerations:

- alignment of data
- proper layout of pages in memory
- cache size
- interaction of the transaction lookaside buffer (TLB) with memory accesses
- combining prefetch and streaming-store instructions.

The guidelines discussed in this chapter come into play in this simple example. TLB priming is required for the Pentium 4 processor just as it is for the Pentium III processor, since software prefetch instructions will not initiate page table walks on either processor.

TLB Priming

The TLB is a fast memory buffer that is used to improve performance of the translation of a virtual memory address to a physical memory address by providing fast access to page table entries. If memory pages are accessed and the page table entry is not resident in the TLB, a TLB miss results and the page table must be read from memory.

The TLB miss results in a performance degradation since another memory access must be performed (assuming that the translation is not already present in the processor caches) to update the TLB. The TLB can be preloaded with the page table entry for the next desired page by accessing (or touching) an address in that page. This is similar to prefetch, but instead of a data cache line the page table entry is being loaded in advance of its use. This helps to ensure that the page table entry is resident in the TLB and that the prefetch happens as requested subsequently.

Optimizing the 8-byte Memory Copy

Example 6-9 presents the copy algorithm that uses second level cache. The algorithm performs the following steps:

1. uses blocking technique to transfer 8-byte data from memory into second-level cache using the `_mm_prefetch` intrinsic, 128 bytes at a time to fill a block. The size of a block should be less than one half of the size of the second-level cache, but large enough to amortize the cost of the loop.
2. loads the data into an `xmm` register using the `_mm_load_ps` intrinsic.
3. transfers the 8-byte data to a different memory location via the `_mm_stream` intrinsics, bypassing the cache. For this operation, it is important to ensure that the page table entry prefetched for the memory is preloaded in the TLB.

Example 6-9 An Optimized 8-byte Memory Copy

```
#define PAGESIZE 4096;
#define NUMPERPAGE 512           // # of elements to fit a page

double a[N], b[N], temp;
for (kk=0; kk<N; kk+=NUMPERPAGE) {
    temp = a[kk+NUMPERPAGE];      // TLB priming
    // use block size = page size,
    // prefetch entire block, one cache line per loop
    for (j=kk+16; j<kk+NUMPERPAGE; j+=16) {
        _mm_prefetch((char*)&a[j], _MM_HINT_NTA);
    }
    // copy 128 byte per loop
    for (j=kk; j<kk+NUMPERPAGE; j+=16) {
        _mm_stream_ps((float*)&b[j],
            _mm_load_ps((float*)&a[j]));
        _mm_stream_ps((float*)&b[j+2],
            _mm_load_ps((float*)&a[j+2]));
    }
}
```

continued

Example 6-9 An Optimized 8-byte Memory Copy

```

    _mm_stream_ps((float*)&b[j+4],
        _mm_load_ps((float*)&a[j+4]));
    _mm_stream_ps((float*)&b[j+6],
        _mm_load_ps((float*)&a[j+6]));
    _mm_stream_ps((float*)&b[j+8],
        _mm_load_ps((float*)&a[j+8]));
    _mm_stream_ps((float*)&b[j+10],
        _mm_load_ps((float*)&a[j+10]));
    _mm_stream_ps((float*)&b[j+12],
        _mm_load_ps((float*)&a[j+12]));
    _mm_stream_ps((float*)&b[j+14],
        _mm_load_ps((float*)&a[j+14]));
}    // finished copying one block
}    // finished copying N elements
_mm_sfence();

```

In Example 6-9, eight `_mm_load_ps` and `_mm_stream_ps` intrinsics are used so that all of the data prefetched (a 128-byte cache line) is written back. The prefetch and streaming-stores are executed in separate loops to minimize the number of transitions between reading and writing data. This significantly improves the bandwidth of the memory accesses.

The instruction, `temp = a[kk+CACHESIZE]`, is used to ensure the page table entry for array, and `a` is entered in the TLB prior to prefetching. This is essentially a prefetch itself, as a cache line is filled from that memory location with this instruction. Hence, the prefetching starts from `kk+4` in this loop.

This example assumes that the destination of the copy is not temporally adjacent to the code. If the copied data is destined to be reused in the near future, then the streaming store instructions should be replaced with regular 128 bit stores (`_mm_store_ps`). This is required because the implementation of streaming stores on Pentium 4 processor writes data directly to memory, maintaining cache coherency.

Application Performance Tools



Intel offers an array of application performance tools that are optimized to take the best advantage of the Intel architecture (IA)-based processors. This appendix introduces these tools and explains their capabilities which you can employ for developing the most efficient programs, without having to write assembly code.

The following performance tools are available:

- **Intel® C++ Compiler and Intel® Fortran Compiler**
The compilers generate highly optimized floating-point code and provide unique features such as profile-guided optimizations and high-level language support, including vectorization, for MMX™ technology, the Streaming SIMD Extensions (SSE), and the Streaming SIMD Extensions 2 (SSE2).
- **Enhanced Debugger**
The Enhanced Debugger (EDB) enables you to debug C++, Fortran or mixed language programs. It allows you to view the XMM registers in a variety of formats corresponding to the data types supported by SSE and SSE2. These registers can also be viewed using the debugger supplied with Microsoft* Visual C++* version 6.0, service pack 4 or later.
- **VTune™ Performance Analyzer**
The VTune analyzer collects, analyzes, and provides Intel architecture-specific software performance data from the system-wide view down to a specific module, function, and instruction in your code.
- **Intel® Performance Library Suite**
The library suite consists of a set of software libraries optimized for Intel architecture processors. The suite currently includes the following libraries:

- Intel® Math Kernel Library (MKL)
- Intel® Signal Processing Library (SPL)
- Intel® Image processing Library (IPL)
- Intel® JPEG library (IJL)
- Intel® Integrated Performance Primitives (IPP)
- Intel® Speech Developer Toolkit (SDT)
- Intel® Recognition Primitives Library (RPL)

Intel Compilers

The Intel C++ compiler is compatible with Microsoft* Visual C++* and plugs in to the Microsoft Developer Studio IDE. The Intel Fortran Compiler can be run out of the Microsoft Developer Studio IDE via the Fortran Build Tool that plugs into it. The Fortran compiler offers substantial source compatibility with Compaq* Visual Fortran.

Both compilers allow you to optimize your code by using special optimization options described in this section. There are several coding methods and optimizations, described here and other sections in this manual, targeted specifically for enabling software developers to optimize applications for the Pentium® III and Intel® Pentium 4 processors. Vectorization, processor dispatch, inter-procedural optimization, and profile-guided optimization are all supported by the Intel compilers and can significantly aid the performance of an application.

The most general optimization options are `-O1` and `-O2`. Each of them enables a number of specific optimization options. In most cases, `-O2` is recommended over `-O1` because the `-O2` option enables inline expansion, which helps programs that have many function calls. The `-O2` option is on by default.

The `-O1` and `-O2` options enable specific options as follows:

- | | |
|------------------|--|
| <code>-O1</code> | Enables options <code>-Og</code> , <code>-Oi</code> , <code>-Os</code> , <code>-Oy</code> , <code>-Ob1</code> , <code>-Gf</code> , <code>-Gs</code> , and <code>-Gy</code> . However, <code>-O1</code> disables a few options that increase code size. |
| <code>-O2</code> | Enables options <code>-Og</code> , <code>-Oi</code> , <code>-Ot</code> , <code>-Oy</code> , <code>-Ob1</code> , <code>-Gf</code> , <code>-Gs</code> , and <code>-Gy</code> . Confines optimizations to the procedural level. |

The `-Od` option disables all optimizations.

All the command-line options are described in the *Intel® C++ Compiler User's Guide and Reference*.

Code Optimization Options

This section describes the options used to optimize your code and improve the performance of your application.

Targeting a Processor (**-Gn**)

Use **-Gn** to target an application to run on a specific processor for maximum performance. Any of the **-Gn** suboptions you choose results in your binary being optimized for corresponding Intel architecture 32-bit processors. **-G6** is the default, and targets optimization for the Pentium II and Pentium III processors. **-G7** targets the Intel Pentium 4 processor.

Automatic Processor Dispatch Support (**-Qx[extensions]** and **-Qax[extensions]**)

The **-Qx[extensions]** and **-Qax[extensions]** options provide support to generate code that is specific to processor-instruction extensions.

-Qx[extensions]	generates specialized code to run exclusively on the processors indicated by the extension(s).
-Qax[extensions]	generates code specialized to processors which support the specified extensions, but also generates generic IA-32 code. The generic code usually executes slower than the specialized version. A runtime check for the processor type is made to determine which code executes.

You can specify the same extensions for either option as follows:

i	Pentium II and Pentium III processors, which use the CMOV and FCMOV instructions
M	Pentium processor with MMX technology, Pentium II, and Pentium III processors
K	Streaming SIMD Extensions. Includes the i and M extensions.
W	Streaming SIMD Extensions 2. Includes the i , M , and K extensions.



CAUTION. When you use `-Qax[extensions]` in conjunction with `-Qx[extensions]`, the extensions specified by `-Qx[extensions]` can be used unconditionally by the compiler, and the resulting program will require the processor extensions to execute properly.

Vectorizer Switch Options

The Intel C++ and Fortran Compiler can vectorize your code using the vectorizer switch options. The options that enable the vectorizer are the `-Qx[M,K,W]` and `-Qax[M,K,W]` described above. The compiler provides a number of other vectorizer switch options that allow you to control vectorization. All vectorization switches require the `-Qx[M,K,W]` or `-Qax[M,K,W]` switch to be on. The default is off.

In addition to the `-Qx[M,K,W]` or `-Qax[M,K,W]` switches, the compiler provides the following vectorization control switch options:

- | | |
|------------------------------|--|
| <code>-Qvec_report[n]</code> | Controls the vectorizer's diagnostic levels, where <code>n</code> is either 0, 1, 2, or 3. |
| <code>-Qrestrict</code> | Enables pointer disambiguation with the <code>restrict</code> qualifier. |

Prefetching

The compilers, with the `-Qx[M,K,W]` and `-Qax[M,K,W]` switches on, insert prefetch instructions, where appropriate, for the Pentium III and Pentium 4 processors.

Loop Unrolling

The compilers automatically unroll loops with the `-Qx[M,K,W]` and `-Qax[M,K,W]` switches.

To disable loop unrolling, specify `-Qunroll0`.

Multithreading with OpenMP

Both the Intel C++ and Fortran Compilers support shared memory parallelism via OpenMP compiler directives, library functions and environment variables. OpenMP directives are activated by the compiler switch `-Qopenmp`. The available directives are described in the Compiler User's Guides available with the Intel C++ and Fortran Compilers, version 5.0 and higher. Further information about the OpenMP standard is available at <http://www.openmp.org>.

Inline Expansion of Library Functions (`-Oi`, `-Oi-`)

The compiler inlines a number of standard C, C++, and math library functions by default. This usually results in faster execution of your program. Sometimes, however, inline expansion of library functions can cause unexpected results. For explanation, see the *Intel® C++ Compiler User's Guide and Reference*.

Floating-point Arithmetic Precision (`-Op`, `-Op-`, `-Qprec`, `-Qprec_div`, `-Qpc`, `-Qlong_double`)

These options provide optimizations with varying degrees of precision in floating-point arithmetic.

Rounding Control Option (`-Qrcd`)

The compiler uses the `-Qrcd` option to improve the performance of code that requires floating-point calculations. The optimization is obtained by controlling the change of the rounding mode.

The `-Qrcd` option disables the change to truncation of the rounding mode in floating-point-to-integer conversions.

For complete details on all of the code optimization options, refer to the *Intel® C++ Compiler User's Guide and Reference*.

Interprocedural and Profile-Guided Optimizations

The following are two methods to improve the performance of your code based on its unique profile and procedural dependencies:

Interprocedural Optimization (IPO)

Use the `-Qip` option to analyze your code and apply optimizations between procedures within each source file. Use multifile IPO with `-Qipo` to enable the optimizations between procedures in separate source files.

Profile-Guided Optimization (PGO)

Creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the compiler generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Profile-guided optimization is particularly beneficial for the Pentium 4 processor. It greatly enhances the optimization decisions the compiler makes regarding instruction cache utilization and memory paging. Also, because PGO uses execution-time information to guide the optimizations, branch-prediction can be significantly enhanced by reordering branches and basic blocks to keep the most commonly used paths in the microarchitecture pipeline, as well as generating the appropriate branch-hints for the processor.

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.



NOTE. *The compiler issues a warning that the dynamic information corresponds to a modified function.*

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

For complete details on the interprocedural and profile-guided optimizations, refer to the *Intel C++ Compiler User's Guide With Support for the Streaming SIMD Extensions 2* (Doc. number 718195-2001).

VTune™ Performance Analyzer

VTune Performance Analyzer is instrumental in helping you understand where to begin tuning your application. VTune analyzer helps you identify and analyze performance trends at all levels: the system, micro-architecture, and application.

The sections that follow discuss the major features of the VTune analyzer that help you improve performance and briefly explain how to use them. For more details on how to sample events, run VTune analyzer and see online help.

Using Sampling Analysis for Optimization

The sampling feature of the VTune analyzer provides analysis of the performance of your applications using time- or event-based sampling and hotspot analysis. The time- or event-based sampling analysis provides the capability to non-intrusively monitor all active software on the system, including the application.

Each sampling session contains summary information about the session, such as the number of samples collected at each privilege level and the type of interrupt used. Each session is associated with a database. The session database allows you to reproduce the results of a session any number of times without having to sample or profile.

Time-based Sampling

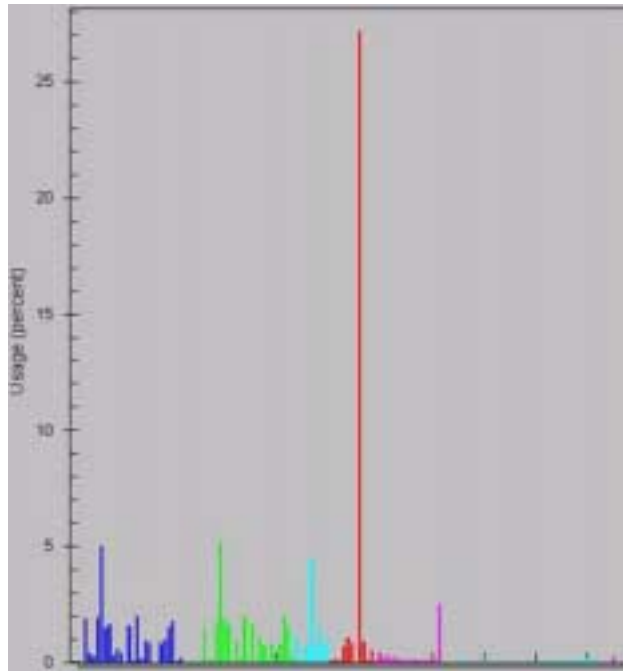
Time-based sampling (TBS) allows you to monitor all active software on your system, including the operating system, device drivers, and application software. TBS collects information at a regular time interval. The VTune analyzer then processes this data to provide a detailed view of the system's activity.

The time-based sampling periodically interrupts the processor at the specified sampling interval and collects samples of the instruction addresses, matches these addresses with an application or an operating system routine, and creates a database with the resulting samples data. VTune analyzer can then graphically display the amount of CPU time spent in each active module, process, and processor (on a multiprocessor system). The TBS—

- samples and display a system-wide view of the CPU time distribution of all the software activity during the sampling session
- determines which sections in your code are taking the most CPU time
- analyzes hotspots, displays the source code, and determines performance issues at the source and assembly code levels.

[Figure A-1](#) provides an example of a hotspots report by location.

Figure A-1 Sampling Analysis of Hotspots by Location



Event-based Sampling

You can use event-based sampling (EBS) to monitor all active software on your system, including the operating system, device drivers, and application software based on the occurrence of processor events.

The VTune analyzer can collect, analyze, and display the performance event counters data of your code provided by the Pentium 4, Pentium III and Pentium II processors. These processors can generate numerous events per clock cycle. The VTune analyzer typically supports programming the events using one of the performance counter.

For event-based sampling, you can select one or more events, in each event group. However, the VTune analyzer runs a separate session to monitor each event you have selected. It interrupts the processor after a specified number of events and collects a sample containing the current instruction address. The frequency at which the samples are collected is determined by how often the event is caused by the software running in the system during the sampling session.

The data collected allows you to determine the number of events that occurred and the impact they had on performance. Sampling results are displayed in the Modules report and Hotspots report. Event data is also available as a performance counter in the Chronologies window. The event sampled per session is listed under the Chronologies entry in the Navigation tree of the VTune analyzer.

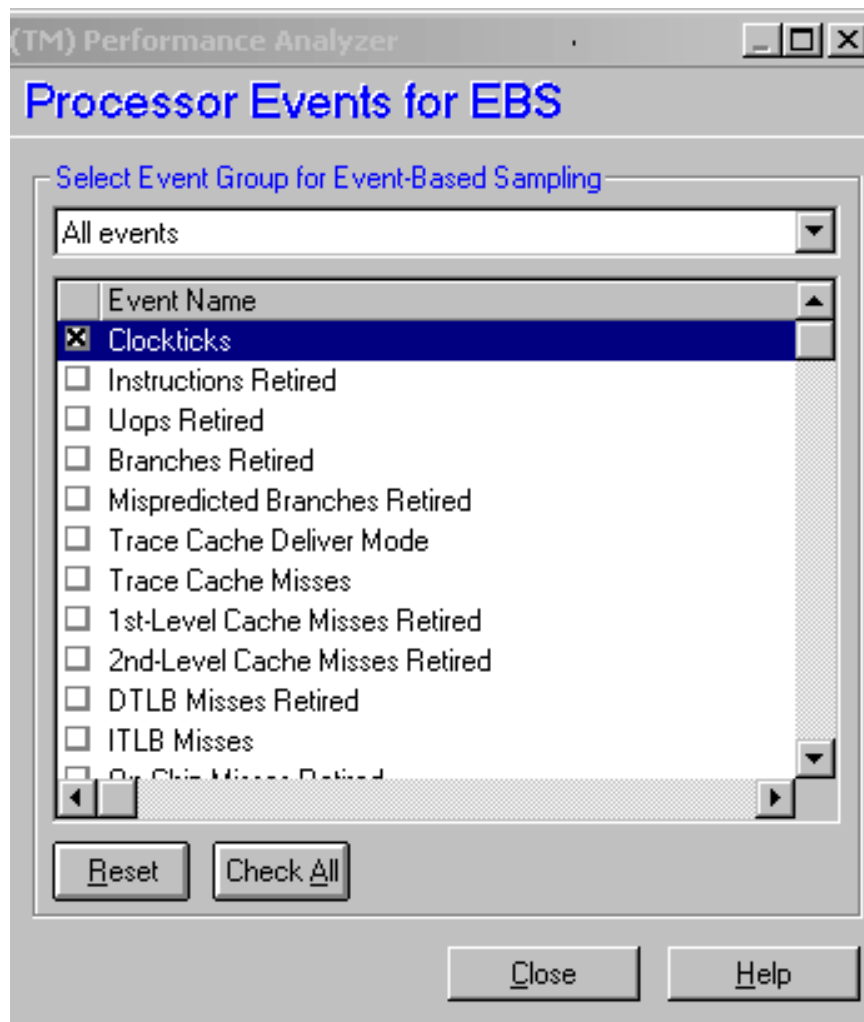
Sampling Performance Counter Events

Event-based sampling can be used together with the hardware performance counters available in the Intel architecture to provide detailed information on the behavior of specific events in the microprocessor. Some of the microprocessor events that can be sampled include instruction retired, branches retired, mispredicted branches retired, trace cache misses.

VTune analyzer provides access to the processor's performance counters. These counters are described in the VTune analysers on-line help documentation, Sampling section. The processors' performance counters can be configured to monitor any of

several different types of events. All the events are listed in the Configure menu/Options command/Processor Events for EBS page of the VTune analyzer, see [Figure A-2](#).

Figure A-2 Processor Events List



At first glance, it is difficult to know which events are relevant for understanding the performance effects. For example, to better understand performance effects on branching and trace cache behavior with the Pentium 4 processor, the VTune analyzer can program various counters to collect the performance data through a variety of pre-defined performance monitoring events. The events that relate to the activity of the Execution Trace cache, as well as the branching include:

- **Branches retired**—this event indicates the number of branch instructions executed to completion.
- **Mispredicted branches retired**—this event indicates the number of mispredicted branch instructions executed to completion.
- **Trace cache deliver mode**—this event indicates the number of cycles that the trace cache is delivering μ ops from the trace cache, Vs. decoding and building traces.
- **Trace cache misses**—this event indicates the number of times that significant delays occurred in order to decode instructions and build a trace because of a trace cache miss.

A complete listing of pre-defined performance monitoring events (also referred to as performance metrics) for the Pentium 4 processor is presented in Appendix B, “Intel Pentium 4 Processor Performance Metrics.” The Pentium 4 processor performance metrics are derived from a set of programmable performance monitoring events. For a list of the programmable performance monitoring events specific to the Pentium 4 processor, refer to Appendix A in the *IA-32 Intel® Architecture Software Developer’s Manual*, Volume 3: System Programming.

Other performance metrics of interest are:

- **Instructions retired**—this event indicates the number of instructions that executed to completion. This does not include partially processed instructions executed due to branch mispredictions.
- **x87 retired**—this event indicates the number of x87 floating-point instructions that executed to completion.
- **Clockticks**—this event initiates time-based sampling by setting the counters to count the processor's clock ticks.

- **x87 Input/Output Assists**—this event indicates the number of occurrence of x87 input/output operands needing assistance to handle an exception condition.

The raw data collected by the VTune analyzer can be used to compute various indicators. For example, ratios of the clockticks, instructions retired, and x87 assists can give you a good indication that the floating-point computation code in the application may be stalled and is suited for re-coding.

Call Graph Profiling

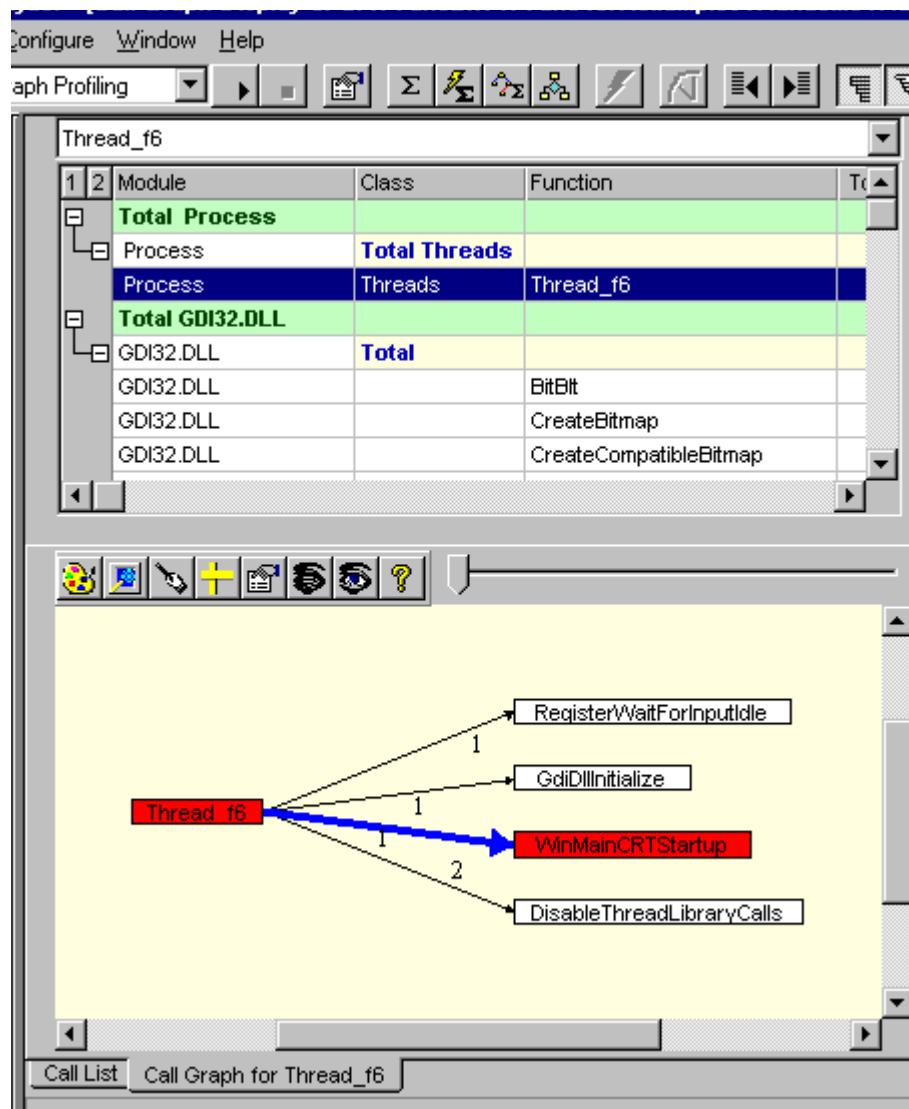
The call graph profiles your application and displays a call graph of active functions. The call graph analyzes the data and displays a graphical view of the threads created during the execution of the application, a complete list of the functions called, and the relationship between the parent and child functions. Use VTune analyzer to profile your Win32* executable files or Java* applications and generate a call graph of active functions.

Call graph profiling includes collecting and analyzing call-site information and displaying the results in the Call List of the Call Graph and Source views. The call graph profiling provides information on how many times a function (caller) called some other function (callee) and the amount of time each call took. In many cases the caller may call the callee from several places (sites), so call graph also provides call information per site. (Call site information is not collected for Java call graphs.)

The View by Call Sites displays the information about callers and callees of the function in question (also referred to as current function) by call sites. This view allows you to locate the most expensive calls.

Call Graph Window

The call graph window comprises three views: Spreadsheet, Call Graph, and Call List, see [Figure A-3](#). The Call Graph view, displayed on the lower section of the window, corresponds to the function (method) selected in the Spreadsheet. It displays the function, the function's parents, and function's child functions.

Figure A-3 Call Graph Window

Each node (box) in the call graph represents a function. Each edge (line with an arrow) connecting two nodes represents the call from the parent (caller) to the child function (callee). The number next to the edge (line) indicates the number of calls to that function.

The window has a Call List tab in the bottom of the Call Graph view. The Call List view lists all the callers and the callees of the function selected in the spreadsheet and displayed in the Call Graph view. In addition, the Call List has a View by Call Sites in which you can see call information represented by call sites.

Static Code Analysis

This feature analyzes performance through

- performing static code analysis of the functions or blocks of code in your application without executing your application
- getting a list of functions with their respective addresses for quick access to your code
- getting summary information about the percentage of pairing and penalties incurred by the instructions in each function.

The static code analyzer provides analysis of the instructions in your application and their relationship with each other, without executing or sampling them. It provides an estimation of the performance of your application, not actual performance. The static code analyzer analyzes the module you specified in the Executable field and displays the results. By default, the static code analyzer analyzes only those functions in the module that have source code available.

During the static code analysis, the static code analyzer does the following tasks:

- searches your program for the debug symbols or prompts you to specify the symbol files
- searches the source directories for the source files
- analyzes each basic block and function in your program
- creates a database with the results

- displays summary information about the performance of each function, including its name, address, the number of instructions executed, the percentage of pairing, the total clock cycles incurred, and the number of clock cycles incurred due to penalties.

Static Assembly Analysis

This feature of the VTune analyzer determines performance issues at the processor level, including the following:

- how many clocks each instruction takes to execute and how many of them were incurred due to penalties
- how your code is executing in the three decode units of the Pentium 4, Pentium III and Pentium II processors.
- regardless of the processor your system is using, the static assembly analyzer analyzes your application's performance as it would run on Intel processors, from Intel486™ to Pentium 4 processors.

The VTune analyzer's static assembly analyzer analyzes basic blocks of code. It assumes that the code and data are already in the cache and ignores loops and jumps. It disassembles your code and displays assembly instructions, annotated with performance information.

The static assembly analyzer disassembles hotspots or static functions in your Windows 95, 98 and NT binary files and analyzes architectural issues that effect their performance. You can invoke Static Assembly Analysis view either by performing a static code analysis or by time or event-based sampling of your binary file. Click on the View Static Assembly Analysis icon in the VTune analyzer's toolbar to view a static analysis of your code and display the assembly view.

Code Coach Optimizations

The code coach performs the following:

- Analyzes C, Fortran, C++, and Java* source code and produces high-level source code optimization advice.
- Analyzes assembly code or disassembled assembly code and produces assembly instruction optimization advice.

Once the VTune analyzer identifies, analyzes, and displays the source code for hotspots or static functions in your application, you can invoke the coach for advice on how to rewrite the code to optimize its performance.

Typically, a compiler is restricted by language pointer semantics when optimizing code. Coach suggests source-level modifications to overcome these and other restrictions. It recognizes commonly used code patterns in your application and suggests how they can be modified to improve performance. The coach window is shown in [Figure A-4](#).

You can invoke the coach from the Source View window by double-clicking on a line of code, or selecting a block of code and then clicking on the code coach icon on the Source View toolbar.

Figure A-4 Code Coach Optimization Advice

There are 2 recommendations identified for the selected code. Double-click on any advice for additional information.

Advice # 1

```

56     void udaxpy (int n, double da, double*dx, double *dy)
57     {
58         if (da == 0.0) return;
59
60         for (int i = 0; i < n; i++)
61             dy[i] = da * dx[i] + dy[i];
62     }
63
64     void udaxpy_vec (int n, double da, double *dx, double *dy)
65     {
66

```

The MULTIPLY operation on line 61 in file C:\Program Files\Intel\iatraining\Samples\w_daxpy_saxpy\saxpy.cpp is a candidate for a performance boost SIMD technology. The following SIMD options are available:

- o **SIMD Class Library:** Using C++, declare the arrays of this statement as objects of the [F64vec2](#) class defined in Intel's SIMD Class Library, and recompile your program using the Intel® C/C++ Compiler. This and other statements like it will be compiled using SIMD code ("vectorized").
- o **Vectorizer:** Use the Intel® C/C++ Compiler vectorizer to automatically generate highly optimized SIMD code. The statement 61 and other statements like it will be vectorized.
- o **Performance Libraries:** Replace your code with calls to functions in the Intel® Performance Library Suite. Some of its functions that are possibly useful in your application appear below (double-click on this advice for a more informative summary):
 - [nspdbMpy3\(\)](#) or [nspdbMPY1\(\)](#)
- o **Intrinsic Functions:** Restructure the loop and use the SIMD intrinsic functions recognized by the Intel® C/C++ Compiler. C-style pseudocode for the intrinsic suggested for this statement (double-click on any function name for a brief description):


```
tmp0 = _mm_mul_pd(_mmset1_pd((double)da), *((__m128d*)&dx[..]));
```

Advice # 2

```

56     void udaxpy (int n, double da, double*dx, double *dy)

```

The coach examines the entire block of code or function you selected and searches for optimization opportunities in the code. As it analyzes your code, it issues error and warning messages much like a compiler parser. Once the coach completes analyzing your code, if it finds suitable optimization advice, it displays the advice in a separate window.

The coach may have more than one advice for a loop or function. If no advice is available, it displays an appropriate message. You can double-click on any advice in the coach window to display context-sensitive help with examples of the original and optimized code.

Where performance can be improved using MMX technology, Streaming SIMD Extensions, or Streaming SIMD Extensions 2 intrinsics, vector classes, or vectorization, the coach provides advice in the form of C-style pseudocode, leaving the data definitions, loop control, and subscripts to the programmer.

For the code using the intrinsics, you can double-click the left mouse button on an argument used in the code to display the description of that argument. Click your right mouse button on an intrinsic to invoke a brief description of that intrinsic.

Assembly Coach Optimization Techniques

Assembly coach uses many optimization techniques to produce its recommended optimized code, for example:

- **Instruction Selection**—assembly coach analyzes each instruction in your code and suggests alternate, equivalent replacements that are faster or more efficient.
- **Instruction Scheduling**—assembly coach uses its in-depth knowledge of processor behavior to suggest an optimal instruction sequence that preserves your code's semantics.
- **Peephole Optimization**—assembly coach identifies particular instruction sequences in your code and replaces them with a single, equivalent instruction.
- **Partial Register Stall Elimination**—assembly coach identifies instruction sequences that can produce partial register stalls and replaces them with alternative sequences that do not cause partial stalls.

In Automatic Optimization and Single Step Optimization modes, you can select or deselect these optimization types in the Assembly Coach Options tab.

Intel® Performance Library Suite

The Intel® Performance Library Suite (PLS) contains a variety of specialized libraries which has been optimized for performance on Intel processors. These optimizations take advantage of appropriate architectural features, including MMX™ technology, Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2). The suite includes the following libraries:

- The Intel® Math Kernel Library: a set of linear algebra, fast Fourier transform functions and vector transcendental functions (Vector Math Library, or VML) for numerical analysis and scientific applications.
- The Intel® Signal Processing Library: set of signal processing functions similar to those available for most Digital Signal Processors (DSPs)
- The Intel® Image Processing Library: a set of low-level image manipulation functions
- The Intel® Integrated Performance Primitives: a cross-platform low-level software-layer integrating functionality across the areas of image and signal processing, speech, computer vision, and audio/video capability. This allows transparent use of IPP across Intel architectures: the full range of IA-32 enhancements, the Itanium™ architecture, the StrongARM® processor, and in the future, the XScale™ micro-architecture.
- The Intel® Speech Developer Toolkit: a set of class libraries and command-line tools for researchers and developers to build complete large vocabulary, speaker independent, speech recognition systems
- The Intel® Recognition Primitives Library: a set of 32-bit recognition primitives for developers of speech- and character-recognition software.

Benefits Summary

The overall benefits the libraries provide to the application developers are as follows:

- Low-level building block functions that support rapid application development, improving time to market
- Highly-optimized routines with a C interface that give Assembly-level performance in a C/C++ development environment (MKL also supports a Fortran interface)

- Processor-specific optimizations that yield the best performance for each Intel processor
- Processor detection and DLL dispatching that loads the appropriate code for the current processor
- Built-in error handling facility that improves productivity in the development cycle

The libraries are optimized for all Intel architecture-based processors, including the Pentium®, Pentium II, Pentium III, Pentium 4, and Itanium™ processors.

Libraries Architecture

Intel Performance Libraries are designed for performance, productivity and ease of use. The Signal Processing Library supports 1D vector-based operations typical of those used in Signal Processing applications. The Image Processing Library supports 2D operations typical of those used for Image Processing applications with appropriate handling of edge effects. The Intel JPEG Library is a small library that supports the compression and decompression of JPEG files and images.

The Math Kernel Library (MKL) is designed for scientific and engineering applications and supports both Fortran and C calling conventions. Its high-performance math functions include, Linear Algebra PACKage (LINPACK), Basic Linear Algebra Subprograms (BLAS) and fast Fourier transforms (FFTs) threaded to run on multiprocessor systems. No change of the code is required for multiprocessor support. The library, including the parts which are not threaded, such as VML, is threadsafe. All libraries employ sophisticated memory management schemes and processor detection.

The Intel Integrated Performance Primitives (IPP) library was derived from the kernel operators initially implemented in Signal, Image, Speech and other real-time functions. IPP functions are light weight kernels without the predefined data structures of other libraries. They are designed for use as building blocks for efficiently constructing and optimizing more complex functions.

Optimizations with the Intel Performance Library Suite

The Intel Performance Library Suite implements a number of optimizations that are discussed throughout this manual. Examples include architecture-specific tuning such as loop unrolling, instruction pairing and scheduling; and memory management with explicit and implicit data prefetching and cache tuning.

The Suite takes advantage of the parallelism in the SIMD instructions using MMX™ technology, Streaming SIMD Extensions (SSE), and Streaming SIMD Extensions 2 (SSE2). These techniques improve the performance of computationally intensive algorithms and deliver hand coded performance in a high level language development environment.

For performance sensitive applications, the Intel Performance Library Suite frees the application developer from the time consuming task of assembly-level programming for a multitude of frequently used functions. The time required for prototyping and implementing new application features is substantially reduced and most important, the time to market is substantially improved. Finally, applications developed with the Intel Performance Library Suite benefit from new architectural features of future generations of Intel processors simply by relinking the application with upgraded versions of the libraries.

Enhanced Debugger (EDB)

The Enhanced Debugger (EDB) enables you to debug C++, Fortran or mixed language programs running under Windows* NT or Windows 2000 (not Windows 98). It allows you to display in a separate window the contents of the eight registers, XMM0 through XMM7, used by the Streaming SIMD Extensions and Streaming SIMD Extensions 2. You may select one of five formats for the register fields: byte (16 bytes); word (8 words); double word (4 double words); single precision (4 single precision floating point); and double precision (2 double precision floating point). When a register is updated, the new value appears in red. The corresponding Streaming SIMD Extensions or Streaming SIMD Extensions 2 instruction can be seen in the disassembly window. For further detail on the features and use of the Enhanced Debugger, refer to the online help.

Intel® Architecture Performance Training Center

The Intel® Architecture Performance Training Center (IAPTC) is a valuable resource for information about Streaming SIMD Extensions 2 (SSE2). For training on how to use the SSE2, refer to the Computer-Based Tutorials (CBTs); for key algorithms and their optimization examples for the Pentium 4 processor, refer to the application notes. You can find information on IAPTC at <http://developer.intel.com/vtune> and additional training information at <http://developer.intel.com/software/idap/training>.

Intel Pentium 4 Processor Performance Metrics



The Intel Pentium 4 processor performance metrics are a set of quantities that are useful for tuning software performance when running applications on the Pentium 4 processor. The metrics are derived from the Pentium 4 processor performance monitoring events, which are described in Chapter 14 and Appendix A of the *IA-32 Intel Architecture Software Developer's Manual*, Volume 3: “System Programming.”

Pentium 4 Processor-Specific Terminology

The descriptions of the Intel Pentium 4 processor performance metrics use Pentium 4 processor-specific terminology presented in the following sections.

Bogus, Non-bogus, Retire

Branch mispredictions incur a large penalty on microprocessors with deep pipelines. In general, the direction of branches can be predicted with a high degree of accuracy by the front end of the Intel Pentium 4 processor, such that most computations can be performed along the predicted path while waiting for the resolution of the branch.

In the event of a misprediction, instructions and micro-ops (μ ops) that were scheduled to execute along the mispredicted path must be cancelled. These instructions and μ ops are referred to as *bogus* instructions and *bogus* μ ops. A number of Pentium 4 processor performance monitoring events, for example, `instruction_retired` and `mops_retired`, can count instructions or μ ops that are retired based on the characterization of bogus versus non-bogus.

In the event descriptions in [Table B-1](#), the term “bogus” refers to instructions or micro-ops that must be cancelled because they are on a path taken from a mispredicted branch. The terms “retired” and “non-bogus” refer to instructions or micro-ops along

the path that results in committed architectural state changes as required by the program execution. Thus instructions and μ ops are either bogus or non-bogus, but not both.

Bus Ratio

Bus Ratio is the ratio of the processor clock to the bus clock. In the Bus Utilization metric, it is the Bus_ratio.

Replay

In order to maximize performance for the common case, the Intel NetBurst micro-architecture sometimes aggressively schedules μ ops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μ ops must be reissued. This mechanism is called replay.

Some occurrences of replays are caused by cache misses, dependence violations (for example, store forwarding problems), and unforeseen resource constraints. In normal operation, some number of replays are common and unavoidable. An excessive number of replays indicate that there is a performance problem.

Assist

When the hardware needs the assistance of microcode to deal with some event, the machine takes an *assist*. One example of such situation is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μ ops before they begin to accumulate, and are costly. The assist mechanism on the Pentium 4 processor is similar in principle to that on the Pentium II processors, which also have an assist event.

Tagging

Tagging is a means of marking μ ops to be counted at retirement. See Appendix A of the *IA-32 Intel Architecture Software Developer's Manual*, Volume 3: "System Programming" for the description of the tagging mechanisms. The same event can happen more than once per μ op. The tagging mechanisms allow a μ op to be tagged once during its lifetime. The retired suffix is used for metrics that increment a count once per μ op, rather than once per event. For example, a μ op may encounter a cache miss more than once during its life time, but a Misses Retired metric (for example, 1st-Level Cache Misses Retired) will increment only once for that μ op.

Metrics Descriptions and Categories

Each performance metric is an expression consisting of one or more event counts, which are collected by programming the Pentium 4 processor performance monitoring events. [Table B-1](#) lists the metrics, and where appropriate, references the performance monitoring events used to form these metrics. The table also describes how to build a metric from the Pentium 4 processor performance monitoring events.

- Column 1 specifies performance metrics. This may be a single-event metric; for example, the metric Instructions Retired is based on the counts of the performance monitoring event `instr_retired`, using a specific set of event mask bits. Or it can be an expression built up from other metrics; for example, IPC is derived from two single-event metrics.
- Column 2 provides a description of the metric in column 1. Please refer to the previous section, ["Pentium 4 Processor-Specific Terminology"](#) for various terms that are specific to the Pentium 4 processor's performance monitoring capabilities.
- Column 3 specifies the performance monitoring event(s) or an algebraic expression(s) that form(s) the metric. There are several metrics that require yet another sub-event in addition to the counting event. The additional sub-event information is included in column 3 as various tags, which are described in section ["Performance Metrics and Tagging Mechanisms"](#). For event names that appear in this column, refer to the *IA-32 Intel Architecture Software Developer's Manual*, Volume 3: "System Programming."

- Column 4 specifies the event mask bit that is needed to use the counting event. The addresses of various model-specific registers (MSR), the event mask bits in Event Select Control registers (ESCR), the bit fields in Counter Configuration Control registers (CCCR) are described in *IA-32 Intel Architecture Software Developer's Manual*, Volume 3: "System Programming."

The metrics listed in Table B-1 are grouped into several categories:

General	Operation not specific to any sub-system of the microarchitecture.
Branching and trace cache	Branching activities and trace cache operation modes
Memory	Memory operation related to the cache hierarch
Bus	Activities related to Front-Side Bus (FSB)
Characterization	Operations specific to the processor core
Machine Clear	Occurrences of severe performance penalties

Table B-1 Pentium 4 Processor Performance Metrics

Metric	Description	Event Name or Metric Expression	Event Mask value required
General metrics			
Clocks	Clockticks	See explanation on how to count clocks in section "Counting Clocks" .	
Instructions Retired	Non-bogus IA-32 instructions executed to completion	Instr_retired	NBOGUSNTAG NBOGUSTAG
IPC	Instructions per cycle	(Instructions Retired) /Clocks	
μops Retired	Non-bogus μops executed to completion	uops_retired	NBOGUS
UPC	μop per cycle	μops Retired/ Clocks	
Branching and Trace Cache (TC) metrics			
Branches Retired	All branch instructions executed to completion	Branch_retired	MMTM MMNM MMTP MMNP
continued			

Table B-1 Pentium 4 Processor Performance Metrics (continued)

Mispredicted Branches Retired	Mispredicted branch instructions executed to completion. This stat is often used in a per-instruction ratio.	Mispred_branch_retired	NBOGUS
Misprediction Ratio	Misprediction rate per branch	(Mispredicted Branches Retired) / (Branches Retired)	
Trace Cache Deliver Mode	The number of cycles that the trace cache is delivering μ ops from the trace cache, vs. decoding and building traces	TC_deliver_mode	DELIVER
% In Deliver Mode	Fraction of all non-sleep cycles that the trace cache is delivering μ ops from vs. decoding and building traces	(Trace Cache Deliver Mode)*100/Clocks	
Trace Cache Misses	The number of times that significant delays occurred in order to decode instructions and build a trace because of a TC miss.	BPU_fetch_request	TCMISS
Memory metrics			
1 st -Level Cache Load Misses Retired	The number of retired μ ops that experienced 1 st -Level cache load misses. This stat is often used in a per-instruction ratio.	Replay_event; set the following replay tag: 1stL_cache_load_miss_retired	NBOGUS
2 nd -Level Cache Load Misses Retired	The number of retired μ ops that experienced 2 nd -Level cache load misses. This stat is often used in a per-instruction ratio.	Replay_event; set the following replay tag: 2ndL_cache_load_miss_retired	NBOGUS
DTLB Load Misses Retired	The number of retired load μ ops that experienced DTLB misses.	Replay_event; set the following replay tag: DTLB_load_miss_retired.	NBOGUS

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

DTLB Store Misses Retired	The number of retired store μ ops that experienced DTLB misses.	Replay_event; set the following replay tag: DTLB_store_miss_retired.	NBOGUS
DTLB Load and Store Misses Retired	The number of retired load or μ ops that experienced DTLB misses. .	Replay_event; set the following replay tag: DTLB_all_miss_retired.	NBOGUS
Page Walk DTLB All Misses	The number of page walk requests due to DTLB misses from either load or store.	page_walk_type	DTMISS
ITLB Misses	The number of ITLB lookups that resulted in a miss. This is more speculative than the actual number of page walks resulted from ITLB misses.	ITLB_reference	MISS
Page Walk Miss ITLB	The number of page walk requests due to ITLB misses.	page_walk_type	ITMISS
Split Load Replays	The number of load references to data that spanned two cache lines.	Memory_complete	LSC
Split Loads Retired	The number of retired load μ ops that spanned two cache lines.	Replay_event; set the following replay tag: Split_load_retired.	NBOGUS
Split Store Replays	The number of store references that spans across cache line boundary.	Memory_complete	SSC
Split Stores Retired	The number of retired store μ ops that spanned two cache lines.	Replay_event; set the following replay tag: Split_store_retired.	NBOGUS
MOB Load Replays	The number of replayed loads related to the Memory Order Buffer (MOB). This metric counts only the case where the store-forwarding data is not an aligned subset of the stored data.	MOB_load_replay	PARTIAL_DATA, UNALGN_ADDR

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

MOB Load Replays Retired	The number of retired load μ ops that experienced replays related to the MOB.	Replay_event; set the following replay tag: MOB_load_replay_retired.	NBOGUS
Loads Retired	The number of retired load operations that were tagged at the front end.	Front_end_event; set the following front end tag: Memory_loads.	NBOGUS
Stores Retired	The number of retired stored operations that were tagged at the front end. This stat is often used in a per-instruction ratio.	Front_end_event; set the following front end tag: Memory_stores.	NBOGUS
Bus metrics			
2 nd -Level Cache Read Misses	The number of 2 nd -level cache read misses	BSQ_2ndL_cache_reference	RD_2L_MISS (Set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
2 nd -Level Cache Read	The number of 2 nd -level cache references	BSQ_2ndL_cache_reference	RD_2L_HITS, RD_2L_HITE, RD_2L_HITM, RD_2L_MISS (Set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
2 nd -Level Cache Miss Ratio	Fraction of 2 nd level reads that miss	(2 nd -Level Cache Miss) / (2 nd -Level Cache Read)	

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

Bus Accesses	The number of all bus transactions that were allocated in the IO Queue	IOQ_allocation	ReqA0, ALL_READ, ALL_WRITE, OWN, PREFETCH (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Non-prefetch Bus Accesses	The number of all bus transactions that were allocated in the IO Queue excluding prefetch	IOQ_allocation	ReqA0, ALL_READ, ALL_WRITE, OWN (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Prefetch Ratio	Fraction of all bus transactions (including retires) that were for HW or SW prefetching.	(Bus Accesses – Nonprefetch Bus Accesses)/ (Bus Accesses)	
FSB Data Ready	The number of front-side bus clocks that the bus is actually being used (including by partials)	FSB_data_activity	DRDY_OWN, DRDY_DRV (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Bus Utilization	The % of time that the bus is actually occupied	(FSB Data Ready) *Bus_ratio*100/ Clocks	

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

Reads	The number of all read transactions on the bus that were allocated in IO Queue	IOQ_allocation	ReqA0, ALL_READ, OWN, PREFETCH (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Writes	The number of all write transactions on the bus that were allocated in IO Queue	IOQ_allocation	ReqA0, ALL_WRITE, OWN (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Reads Non-prefetch	The number of all read transactions on the bus that originated from the processor.	IOQ_allocation	ReqA0, ALL_WRITE, OWN (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

All WC	The number of Write-Combining (WC) memory transactions on the bus that originated from the processor.	IOQ_allocation	ReqA0, MEM_WC, OWN (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
All UC	The number of UC (Uncacheable) memory transactions on the bus that originated from the processor.	IOQ_allocation	ReqA0, MEM_UC, OWN (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Write WC Full	The number of full-size write (but not writeback) transactions of the WC type on the bus.	BSQ_allocation	REQ_TYPE1 REQ_LEN0 REQ_LEN1 MEM — TYPE0 REQ_DEM_TYPE (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)

continued

Table B-1 Pentium 4 Processor Performance Metrics (continued)

Write WC Partial	The number of partial write (but not writeback) transactions of the WC type on the bus.	BSQ_allocation	REQ_TYPE1 REQ_LEN0 MEM_TYPE0 REQ_DEM_TYPE (Also set the following CCCR bits to make edge triggered: Compare=1; Edge=1; Threshold=0)
Characterization metrics			
x87 Input Assists	The number of occurrences of x87 input operands needing assistance to handle an exception condition. This stat is often used in a per-instruction ratio.	X87_assists	PREA
x87 Output Assists	The number of occurrences of x87 operations needing assistance to handle an exception condition.	X87_assists	POAO, POAU
SSE Input Assists	The number of occurrences of SSE/SSE2 floating-point operations needing assistance to handle an exception condition. The number of occurent includes speculative counts.	SSE_input_assist	ALL
Packed SP Retired ¹	Non-bogus packed single-precision instructions retired	Execution_event; set this execution tag: Packed_SP_retired	NONBOGUS0
Packed DP Retired ¹	Non-bogus packed double-precision instructions retired	Execution_event; set this execution tag: Packed_DP_retired	NONBOGUS0
continued			

Table B-1 Pentium 4 Processor Performance Metrics (continued)

Scalar SP Retired ¹	Non-bogus scalar single-precision instructions retired	Execution_event; set this execution tag: Scalar_SP_retired	NONBOGUS0
Scalar DP Retired ¹	Non-bogus scalar double-precision instructions retired	Execution_event; set this execution tag: Scalar_DP_retired	NONBOGUS0
64-bit MMX Instructions Retired ¹	Non-bogus 64-bit integer SIMD instruction (MMX instructions) retired	Execution_event; set the following execution tag: 64_bit_MMX_retired	NONBOGUS0
128-bit MMX Instructions Retired ¹	Non-bogus 128-bit integer SIMD instructions retired	Execution_event; set this execution tag: 128_bit_MMX_retired	NONBOGUS0
X87 Retired ²	Non-bogus x87 floating-point instructions retired	Execution_event; set this execution tag: X87_FP_retired	NONBOGUS0
x87 SIMD Memory Moves Retired ³	Non-bogus x87 and SIMD memory operation and move instructions retired	Execution_event; set this execution tag: X87_SIMD_memory_moves_retired	NONBOGUS0
Machine clear metrics			
Machine Clear Count	The number of cycles that the entire pipeline of the machine is cleared for all causes.	Machine_clear	CLEAR (Also Set the following CCCR bits: Compare=1; Edge=1; Threshold=0)
Memory Order Machine Clear	The number of times that the entire pipeline of the machine is cleared due to memory-ordering issues.	Machine_clear	MOCLEAR

1. Most MMX technology instructions, Streaming SIMD Extensions and Streaming SIMD Extensions 2 decode into a single μ op. There are some instructions that decode into several μ ops; in these limited cases, the metrics count the number of μ ops that are actually tagged.
2. Most commonly used x87 instructions (e.g. `fmul`, `fadd`, `fdiv`, `fsqrt`, `fstp`, etc...) decode into a single μ op. However, transcendental and some x87 instructions decode into several μ ops; in these limited cases, the metrics will count the number of μ ops that are actually tagged.

3. Load and store operations, register-to-register moves for x87 floating-point instructions, MMX™ technology instructions, Streaming SIMD Extensions, and Streaming SIMD Extensions 2 are included in this metric. Load and store operations, as well as register-to-register moves for integer instruction are not included in this metric. Some instructions decode into several memory/moves μ ops; for example, `movdqu` contains two separate 64-bit data store operations; in these cases, the metrics count all memory/moves μ ops that are actually tagged.

Performance Metrics and Tagging Mechanisms

A number of metrics require more tags to be specified in addition to programming a counting event; for example, the metric Split Loads Retired requires specifying a `split_load_retired` tag in addition to programming the `replay_event` to count at retirement. This section describes three sets of tags that are used in conjunction with three at-retirement counting events: `front_end_event`, `replay_event`, and `execution_event`. Please refer to Appendix A of the “IA-32 Intel® Architecture Software Developer’s Manual, Volume 3: System Programming” for the description of the at-retirement events.

Tags for `replay_event`

[Table B-2](#) provides a list of the tags that are used by various metrics in [Table B-1](#). These tags enable you to mark μ ops at earlier stage of execution and count the μ ops at retirement using the `replay_event`. These tags require at least two MSR’s (see [Table B-2](#), column 2 and column 3) to tag the μ ops so they can be detected at retirement. Some tags require additional MSR (see [Table B-2](#), column 4) to select the event types for these tagged μ ops. The event names referenced in column 4 are those from the Pentium 4 processor performance monitoring events.

Table B-2 Metrics That Utilize Replay Tagging Mechanism

Replay Metric Tags ¹	Bit field to set: IA32_PEBS_ ENABLE	Bit field to set: MSR_PEBS_ MATRIX_VERT	Additional MSR	See Event Mask Parameter for Replay_event
1stL_cache_load _miss_retired	Bit 0, BIT 24, BIT 25	Bit 0	None	NBOGUS
2ndL_cache_load _miss_retired	Bit 1, BIT 24, BIT 25	Bit 0	None	NBOGUS

continued

Table B-2 Metrics That Utilize Replay Tagging Mechanism (continued)

DTLB_load_miss_retired	Bit 2, BIT 24, BIT 25	Bit 0	None	NBOGUS
DTLB_store_miss_retired	Bit 2, BIT 24, BIT 25	Bit 1	None	NBOGUS
DTLB_all_miss_retired	Bit 2, BIT 24, BIT 25	Bit 0, Bit 1	None	NBOGUS
MOB_load_replay_retired	Bit 9, BIT 24, BIT 25	Bit 0	Select MOB_load_replay and set the PARTIAL_DATA and UNALGN_ADDR bits	NBOGUS
Split_load_retired	Bit 10, BIT 24, BIT 25	Bit 0	Select Load_port_replay event on SAAT_CR_ESCR1 and set SPLIT_LD bit	NBOGUS
Split_store_retired	Bit 10, BIT 24, BIT 25	Bit 1	Select Store_port_replay event on SAAT_CR_ESCR0 and set SPLIT_ST bit	NBOGUS

1. Certain kinds of μ ops cannot be tagged. These include I/O operations, UC and locked accesses, returns, and far transfers.

Tags for front_end_event

[Table B-3](#) provides a list of the tags that are used by various metrics derived from the front_end_event. The event names referenced in column 2 can be found from the Pentium 4 processor performance monitoring events.

Table B-3 Table 3 Metrics that utilize the front-end tagging mechanism

Front-end MetricTags ¹	Additional MSR	See Event Mask Parameter for Front_end_event
Memory_loads	Set the TAGLOADS bit in Uop_Type	NBOGUS
Memory_stores	Set the TAGSTORES bit in Uop_Type	NBOGUS

1. There may be some undercounting of front end events when there is an overflow or underflow of the floating point stack.

Tags for execution_event

[Table B-4](#) provides a list of the tags that are used by various metrics derived from the execution_event. These tags require programming an upstream ESCR to select event mask with its TagUop and TagValue bit fields. The event mask for the downstream ESCR is specified in column 4. The event names referenced in column 4 can be found in the Pentium 4 processor performance monitoring events.

Table B-4 Metrics that utilize the execution tagging mechanism

Execution Metric Tags	Upstream ESCR	Tag Value in Upstream ESCR	See Event Mask Parameter for Execution_event
Packed_SP_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of packed_SP_uop,	1	NBOGUS0
Packed_DP_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of packed_DP_uop,	1	NBOGUS0
Scalar_SP_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of scalar_SP_uop,	1	NBOGUS0
Scalar_DP_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of scalar_DP_uop,	1	NBOGUS0
128_bit_MMX_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of 128_bit_MMX_uop,	1	NBOGUS0
64_bit_MMX_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of 64_bit_MMX_uop,	1	NBOGUS0
X87_FP_retired	Set the ALL bit in the event mask and the TagUop bit in the ESCR of x87_FP_uop,	1	NBOGUS0
X87_SIMD_memory_moves_retired	Set the ALLP0 and ALLP2 bits in event mask and the TagUop bit in the ESCR of X87_SIMD_moves_uop,	1	NBOGUS0

Counting Clocks

This section describes two ways to count processor clocks.

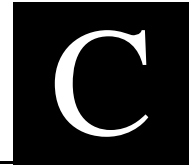
The time stamp counter increments whenever the chip is not in deep-sleep mode. It can be read with the RDTSC instruction. The difference in values between two reads (modulo 2^{64}) gives the number of processor clocks between those reads.

The performance monitoring counters can also be configured to count clocks whenever the chip is not in deep sleep mode. To count clocks with a performance monitoring counter, do the following:

- Select any one of the 18 counters.
- Select any of the possible ESCRs whose events the selected counter can count, and set its event select to anything other than `no_event`. This may not seem necessary, but the counter may be disabled in some cases if this is not done.
- Turn threshold comparison on in the CCCR by setting the compare bit to 1.
- Set the threshold to 15 and the complement to 1 in the CCCR. Since no event can ever exceed this threshold, the threshold condition is met every cycle, and hence the counter counts every cycle.
- Enable counting in the CCCR for that counter by setting the enable bit.

The second alternative for counting clocks is useful where counter overflow is used to generate an interrupt and for those cases where it is easier for a tool to read a performance counter instead of the time stamp counter.

IA-32 Instruction Latency and Throughput



This appendix contains tables of the latency, throughput and execution units that are associated with IA-32 instructions. The data in these tables are specific to the Intel Pentium 4 processor. For detailed discussions of the Intel NetBurst micro-architecture and the relevance of instruction throughput and latency information for code tuning, see [“Execution Core Detail” in Chapter 1](#) and [“Floating Point/SIMD Operands” in Chapter 2](#).

This appendix contains the following sections:

- “Overview” – an overview of issues related to instruction selection and scheduling.
- “Definitions” – the definitions for the primary information presented in the tables in section “Latency and Throughput.”
- “Latency and Throughput” – the listings of IA-32 instruction throughput, latency and execution units associated with each instruction.

Overview

The Pentium 4 processor uses out-of-order execution with dynamic scheduling and buffering to tolerate poor instruction selection and scheduling that may occur in legacy code. It can reorder μ ops to cover latency delays and to avoid resource conflicts. In some cases, the micro-architecture’s ability to avoid such delays can be enhanced by arranging IA-32 instructions. While reordering IA-32 instructions may help, the execution core determines the final schedule of μ ops.

This appendix provides information to assembly language programmers and compiler writers, to aid in selecting the sequence of instructions which minimizes dependence chain latency, and to arrange instructions in an order which assists the hardware in processing instructions efficiently while avoiding resource conflicts. The performance

impact of applying the information presented in this appendix has been shown to be on the order of several percent, for applications which are not completely dominated by other performance factors, such as:

- cache miss latencies
- bus bandwidth
- I/O bandwidth.

Instruction selection and scheduling matters most when the compiler or assembly programmer has already addressed the performance issues discussed in Chapter 2:

- observe store forwarding restrictions
- avoid cache line and memory order buffer splits
- do not inhibit branch prediction.
- minimize the use of `xchg` instructions on memory locations

While several items on the above list involve selecting the right instruction, this appendix focuses on the following issues. These are listed in an expected priority order, though which item contributes most to performance will vary by application.

- Maximize the flow of μ ops into the execution core. IA-32 instructions which consist of more than four μ ops are executed from microcode ROM. These instructions with longer μ op flows incur a slight overhead for switching between the execution trace cache and the microcode ROM. Transfers to microcode ROM often reduce how efficiently μ ops can be packed into the trace cache. Where possible, it is advisable to select instructions with four or fewer μ ops. For example, a 32-bit integer multiply with a memory operand fits in the trace cache without going to microcode, while a 16-bit integer multiply to memory does not.
- Avoid resource conflicts. Interleaving instructions so that they don't compete for the same port or execution unit can increase throughput. For example, alternating `PADDQ` and `PMULUDQ`, each have a throughput of one issue per two clock cycles. When interleaved, they can achieve an effective throughput of one instruction per cycle because they use the same port but different execution units. Selecting instructions with fast throughput also helps to preserve issue port bandwidth, hide latency and allows for higher software performance.

- Minimize the latency of dependence chains that are on the critical path. For example, an operation to shift left by two bits executes faster when encoded as two adds than when it is encoded as a shift. If latency is not an issue, the shift results in a denser byte encoding.

In addition to the general and specific rules, coding guidelines and the instruction data provided in this manual, you can take advantage of the software performance analysis and tuning toolset available at <http://developer.intel.com/software/products/index.htm>. The tools include the VTune Performance Analyzer, with its performance-monitoring capabilities, Intel® Graphics Performance Toolkit, and a few others.

Definitions

The IA-32 instruction performance data are listed in several tables. The tables contain the following information:

Instruction Name: The assembly mnemonic of each instruction.

Latency: The number of clock cycles that are required for the execution core to complete the execution of all of the μ ops that form a IA-32 instruction.

Throughput: The number of clock cycles required to wait before the issue ports are free to accept the same instruction again. For many IA-32 instructions, the throughput of an instruction can be significantly less than its latency.

Execution units: The names of the execution units in the execution core that are utilized to execute the μ ops for each instruction. This information is provided only for IA-32 instructions that are decoded into no more than 4 μ ops. μ ops for instructions that decode into more than 4 μ ops are supplied by microcode ROM. Note that several execution units may share the same port, such as FP_ADD, FP_MUL, or MMX_SHFT in the FP_EXECUTE cluster (see [Figure 1-4](#)).

Latency and Throughput

This section presents the latency and throughput information for the IA-32 instruction set including the Streaming SIMD Extensions 2, Streaming SIMD Extensions, MMX technology, and most of the frequently used general-purpose integer and x87 floating-point instructions.

Due to the complexity of dynamic execution and out-of-order nature of the execution core, the instruction latency data may not be sufficient to accurately predict realistic performance of actual code sequences based on adding instruction latency data.

- The instruction latency data are only meant to provide a relative comparison of instruction-level performance of IA-32 instructions based on the Intel NetBurst micro-architecture.
- All numeric data in the tables are:
 - approximate and are subject to change in future implementations of the Intel NetBurst micro-architecture.
 - not meant to be used as reference numbers for comparisons of instruction-level performance benchmarks. Comparison of instruction-level performance of microprocessors that are based on different micro-architecture is a complex subject that requires additional information that is beyond the scope of this manual.

Comparisons of latency and throughput data between the Pentium 4 processor and the Pentium III processor can be misleading, because one cycle in the Pentium 4 processor is NOT equal to one cycle in the Pentium III processor. The Pentium 4 processor is designed to operate at higher clock frequencies than the Pentium III processor. Many IA-32 instructions can operate with either registers as their operands or with a combination of register/memory address as their operands. The performance of a given instruction between these two types is different.

The section that follows, [“Latency and Throughput with Register Operands”](#), gives the latency and throughput data for the register-to-register instruction type. Section [“Latency and Throughput with Memory Operands”](#) discusses how to adjust latency and throughput specifications for the register-to-memory and memory-to-register instructions.

In some cases, the latency or throughput figures given are just one half of a clock. This occurs only for the double-speed ALUs.

Latency and Throughput with Register Operands

The IA-32 instruction latency and throughput data are presented in [Table C-1](#) through [Table C-7](#). The tables include all instructions of the Streaming SIMD Extension 2, Streaming SIMD Extension, MMX technology and most of the commonly used IA-32 instructions.

Table C-1 Streaming SIMD Extension 2 128-bit Integer Instructions

Instruction	Latency ¹	Throughput	Execution Unit ²
CVTDQ2PS ³ xmm, xmm	5	2	FP_ADD
CVTPS2DQ ³ xmm, xmm	5	2	FP_ADD
CVTTPS2DQ ³ xmm, xmm	5	2	FP_ADD
MOVD xmm, r32	6	2	MMX_MISC,MMX_SHFT
MOVD r32, xmm	10	1	FP_MOVE,FP_MISC
MOVDQA xmm, xmm	6	1	FP_MOVE
MOVDQU xmm, xmm	6	1	FP_MOVE
MOVDQ2Q mm, xmm	8	2	FP_MOVE,MMX_ALU
MOVQ2DQ xmm, mm	8	2	FP_MOVE,MMX_SHFT
MOVQ xmm, xmm	2	2	MMX_SHFT
PACKSSWB/PACKSSDW/ PACKUSWB xmm, xmm	4	2	MMX_SHFT
PADDB/PADDW/PADDD xmm, xmm	2	2	MMX_ALU
PADDSB/PADDSW/ PADDUSB/PADDUSW xmm, xmm	2	2	MMX_ALU
PADDQ/PSUBQ mm, mm	2	1	MMX_ALU
PADDQ/ PSUBQ ³ xmm, xmm	6	2	MMX_ALU
PAND xmm, xmm	2	2	MMX_ALU
PANDN xmm, xmm	2	2	MMX_ALU
PAVGB/PAVGW xmm, xmm	2	2	MMX_ALU
PCMPEQB/PCMPEQD/ PCMPEQW xmm, xmm	2	2	MMX_ALU
PCMPGTB/PCMPGTD/PCMPGTW xmm, xmm	2	2	MMX_ALU
continued			

Table C-1 Streaming SIMD Extension 2 128-bit Integer Instructions (continued)

Instruction	Latency ¹	Throughput	Execution Unit ²
PEXTRW r32, xmm, imm8	7	2	MMX_SHFT,FP_MISC
PINSRW xmm, r32, imm8	4	2	MMX_SHFT,MMX_MISC
PMADDWD xmm, xmm	8	2	FP_MUL
PMAX xmm, xmm	2	2	MMX_ALU
PMIN xmm, xmm	2	2	MMX_ALU
PMOVMKKB ³ r32, xmm	7	2	FP_MISC
PMULHUW/PMULHW/ PMULLW ³ xmm, xmm	8	2	FP_MUL
PMULUDQ mm, mm	8	2	FP_MUL
POR xmm, xmm	2	2	MMX_ALU
PSADBW xmm, xmm	4	2	MMX_ALU
PSHUFD xmm, xmm, imm8	4	2	MMX_SHFT
PSHUFHW xmm, xmm, imm8	2	2	MMX_SHFT
PSHUFLW xmm, xmm, imm8	2	2	MMX_SHFT
PSLLDQ xmm, imm8	4	2	MMX_SHFT
PSLLW/PSLLD/PSLLQ xmm, xmm/imm8	2	2	MMX_SHFT
PSRAW/PSRAD xmm, xmm/imm8	2	2	MMX_SHFT
PSRLDQ xmm, imm8	4	2	MMX_SHFT
PSRLW/PSRLD/PSRLQ xmm, xmm/imm8	2	2	MMX_SHFT
PSUBB/PSUBW/PSUBD xmm, xmm	2	2	MMX_ALU
PSUBSB/PSUBSW/PSUBUSB/PSUBUS W xmm, xmm	2	2	MMX_ALU
PUNPCKHBW/PUNPCKHWD/PUNPCK HDQ/PUNPCKHQDQ xmm, xmm	4	2	MMX_SHFT
PUNPCKLBW/PUNPCKLWD/PUNPCKL DQ xmm, xmm	2	2	MMX_SHFT
PUNPCKLQDQ ³ xmm, xmm	4	1	FP_MISC
PXOR xmm, xmm	2	2	MMX_ALU

See [“Table Footnotes”](#)

Table C-2 Streaming SIMD Extension 2 Double-precision Floating-point Instructions

Instruction	Latency ¹	Throughput	Execution Unit ²
ADDPD xmm, xmm	4	2	FP_ADD
ADDSD xmm, xmm	4	2	FP_ADD
ANDNPD ³ xmm, xmm	4	2	MMX_ALU
ANDPD ³ xmm, xmm	4	2	MMX_ALU
CMPPD xmm, xmm	4	2	FP_ADD
CMPSD xmm, xmm, imm8	4	2	FP_ADD
COMISD xmm, xmm	6	2	FP_ADD, FP_MISC
CVTDQ2PD xmm, xmm	8	3	FP_ADD, MMX_SHFT
CVTPD2PI mm, xmm	11	3	FP_ADD, MMX_SHFT,MMX_ALU
CVTPD2DQ xmm, xmm	9	2	FP_ADD, MMX_SHFT
CVTPD2PS ³ xmm, xmm	10	2	FP_ADD, MMX_SHFT
CVTPI2PD xmm, mm	11	4	FP_ADD, MMX_SHFT,MMX_ALU
CVTPS2PD ³ xmm, xmm	10	4	FP_ADD, MMX_SHFT,MMX_ALU
CVTSD2SI r32, xmm	8	2	FP_ADD, FP_MISC
CVTSD2SS ³ xmm, xmm	16	4	FP_ADD, MMX_SHFT
CVTSI2SD ³ xmm, r32	15	3	FP_ADD, MMX_SHFT, MMX_MISC
CVTSS2SD ³ xmm, xmm	14	3	
CVTTPD2PI mm, xmm	11	3	FP_ADD, MMX_SHFT,MMX_ALU
CVTTPD2DQ xmm, xmm	9	2	FP_ADD, MMX_SHFT
CVTTSD2SI r32, xmm	8	2	FP_ADD, FP_MISC
DIVPD xmm, xmm	62	62	FP_DIV
DIVSD xmm, xmm	35	35	FP_DIV
MAXPD xmm, xmm	4	2	FP_ADD
MAXSD xmm, xmm	4	2	FP_ADD
MINPD xmm, xmm	4	2	FP_ADD
MINSD xmm, xmm	4	2	FP_ADD
MOVAPD xmm, xmm	6	1	FP_MOVE
MOVMSKPD r32, xmm	6	2	FP_MISC

continued

Table C-2 Streaming SIMD Extension 2 Double-precision Floating-point Instructions (continued)

Instruction	Latency ¹	Throughput	Execution Unit ²
MOVSD xmm, xmm	6	2	MMX_SHFT
MOVUPD xmm, xmm	6	1	FP_MOVE
MULPD xmm, xmm	6	2	FP_MUL
MULSS xmm, xmm	6	2	FP_MUL
ORPD ³ xmm, xmm	4	2	MMX_ALU
SHUFPD ³ xmm, xmm, imm8	6	2	MMX_SHFT
SQRTPD xmm, xmm	62	62	FP_DIV
SQRTSD xmm, xmm	35	35	FP_DIV
SUBPD xmm, xmm	4	2	FP_ADD
SUBSD xmm, xmm	4	2	FP_ADD
UCOMISD xmm, xmm	6	2	FP_ADD, FP_MISC
UNPCKHPD ³ xmm, xmm	6	2	MMX_SHFT
UNPCKLPD ³ xmm, xmm	4	2	MMX_SHFT
XORPD ³ xmm, xmm	4	2	MMX_ALU

See [“Table Footnotes”](#)**Table C-3 Streaming SIMD Extension Single-precision Floating-point Instructions**

Instruction	Latency ¹	Throughput	Execution Unit ²
ADDPS xmm, xmm	4	2	FP_ADD
ADDSS xmm, xmm	4	2	FP_ADD
ANDNPS ³ xmm, xmm	4	2	MMX_ALU
ANDPS ³ xmm, xmm	4	2	MMX_ALU
CMPPS xmm, xmm	4	2	FP_ADD
CMPSS xmm, xmm	4	2	FP_ADD
COMISS xmm, xmm	6	2	FP_ADD, FP_MISC
CVTPI2PS xmm, mm	11	4	MMX_ALU, FP_ADD, MMX_SHFT

continued

Table C-3 Streaming SIMD Extension Single-precision Floating-point Instructions (continued)

Instruction	Latency ¹	Throughput	Execution Unit ²
CVTPS2PI mm, xmm	7	2	FP_ADD,MMX_ALU
CVTSI2SS ³ xmm, r32	11	2	FP_ADD,MMX_SHFT, MMX_MISC
CVTSS2SI r32, xmm	8	2	FP_ADD,FP_MISC
CVTTPS2PI mm, xmm	7	2	FP_ADD,MMX_ALU
CVTTSS2SI r32, xmm	8	2	FP_ADD,FP_MISC
DIVPS xmm, xmm	32	32	FP_DIV
DIVSS xmm, xmm	22	22	FP_DIV
MAXPS xmm, xmm	4	2	FP_ADD
MAXSS xmm, xmm	4	2	FP_ADD
MINPS xmm, xmm	4	2	FP_ADD
MINSS xmm, xmm	4	2	FP_ADD
MOVAPS xmm, xmm	6	1	FP_MOVE
MOVHLPS ³ xmm, xmm	6	2	MMX_SHFT
MOVLHPS ³ xmm, xmm	4	2	MMX_SHFT
MOVMSKPS r32, xmm	6	2	FP_MISC
MOVSS xmm, xmm	4	2	MMX_SHFT
MOVUPS xmm, xmm	6	1	FP_MOVE
MULPS xmm, xmm	6	2	FP_MUL
MULSS xmm, xmm	6	2	FP_MUL
ORPS ³ xmm, xmm	4	2	MMX_ALU
RCPPS ³ xmm, xmm	6	4	MMX_MISC
RCPSS ³ xmm, xmm	6	2	MMX_MISC,MMX_SHFT
RSQRTPS ³ xmm, xmm	6	4	MMX_MISC
RSQRTSS ³ xmm, xmm	6	4	MMX_MISC,MMX_SHFT
SHUFPS ³ xmm, xmm, imm8	6	2	MMX_SHFT
SQRTPS xmm, xmm	32	32	FP_DIV
SQRTSS xmm, xmm	22	22	FP_DIV
SUBPS xmm, xmm	4	2	FP_ADD

continued

Table C-3 Streaming SIMD Extension Single-precision Floating-point Instructions (continued)

Instruction	Latency ¹	Throughput	Execution Unit ²
SUBSS xmm, xmm	4	2	FP_ADD
UCOMISS xmm, xmm	6	2	FP_ADD, FP_MISC
UNPCKHPS ³ xmm, xmm	6	2	MMX_SHFT
UNPCKLPS ³ xmm, xmm	4	2	MMX_SHFT
XORPS ³ xmm, xmm	4	2	MMX_ALU

See [“Table Footnotes”](#)**Table C-4 Streaming SIMD Extension 64-bit Integer Instructions**

Instruction	Latency ¹	Throughput	Execution Unit
PAVGB/PAVGW mm, mm	2	1	MMX_ALU
PEXTRW r32, mm, imm8	7	2	MMX_SHFT, FP_MISC
PINSRW mm, r32, imm8	4	1	MMX_SHFT, MMX_MISC
PMAX mm, mm	2	1	MMX_ALU
PMIN mm, mm	2	1	MMX_ALU
PMOVMASKB ³ r32, mm	7	2	FP_MISC
PMULHUW ³ mm, mm	8	1	FP_MUL
PSADBW mm, mm	4	1	MMX_ALU
PSHUFW mm, mm, imm8	2	1	MMX_SHFT

See [“Table Footnotes”](#)

Table C-5 MMX™ Technology 64-bit Instructions

Instruction	Latency ¹	Throughput	Execution Unit ²
MOVD mm, r32	2	1	MMX_ALU
MOVD ³ r32, mm	5	1	FP_MISC
MOVQ mm, mm	6	1	FP_MOV
PACKSSWB/PACKSSDW/PACKUSWB mm, mm	2	1	MMX_SHFT
PADDB/PADDW/PADDD mm, mm	2	1	MMX_ALU
PADDSB/PADDSW/PADDUSB/PADDUSW mm, mm	2	1	MMX_ALU
PAND mm, mm	2	1	MMX_ALU
PANDN mm, mm	2	1	MMX_ALU
PCMPEQB/PCMPEQD/PCMPEQW mm, mm	2	1	MMX_ALU
PCMPGTB/PCMPGTD/PCMPGTW mm, mm	2	1	MMX_ALU
PMADDWD ³ mm, mm	8	1	FP_MUL
PMULHW/PMULLW ³ mm, mm	8	1	FP_MUL
POR mm, mm	2	1	MMX_ALU
PSLLQ/PSLLW/PSLLD mm, mm/imm8	2	1	MMX_SHFT
PSRAW/PSRAD mm, mm/imm8	2	1	MMX_SHFT
PSRLQ/PSRLW/PSRLD mm, mm/imm8	2	1	MMX_SHFT
PSUBB/PSUBW/PSUBD mm, mm	2	1	MMX_ALU
PSUBSB/PSUBSW/PSUBUSB/PSUBUSW mm, mm	2	1	MMX_ALU
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ mm, mm	2	1	MMX_SHFT
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ mm, mm	2	1	MMX_SHFT
PXOR mm, mm	2	1	MMX_ALU
EMMS ¹	12	12	

See [“Table Footnotes”](#)

Table C-6 IA-32 x87 Floating-point Instructions

Instruction	Latency ¹	Throughput	Execution Unit ²
FABS	2	1	FP_MISC
FADD	5	1	FP_ADD
FSUB	5	1	FP_ADD
FMUL	7	2	FP_MUL
FCOM	2	1	FP_MISC
FCHS	2	1	FP_MISC
FDIV Single Precision	23	23	FP_DIV
FDIV Double Precision	38	38	FP_DIV
FDIV Extended Precision	43	43	FP_DIV
FSQRT SP	23	23	FP_DIV
FSQRT DP	38	38	FP_DIV
FSQRT EP	43	43	FP_DIV
F2XM1 ⁴	90-150	60	
FCOS ⁴	190-240	130	
FPATAN ⁴	150-300	140	
FPTAN ⁴	225-250	170	
FSIN ⁴	160-180	130	
FSINCOS ⁴	160-220	140	
FYL2X ⁴	140-190	85	
FYL2XP1 ⁴	140-190	85	
FSCALE ⁴	60	7	
FRNDINT ⁴	30	11	
FXCH ⁵	0	1	FP_MOVE

See [“Table Footnotes”](#)

Table C-7 IA-32 General Purpose Instructions

Instruction	Latency ¹	Throughput	Execution Unit ²
ADC/SBB reg, reg	8	3	
ADC/SBB reg, imm	6	2	ALU
ADD/SUB	0.5	0.5	ALU
AND/OR/XOR	0.5	0.5	ALU
CMP/TEST	0.5	0.5	ALU
DEC/INC	1	0.5	ALU
IMUL r32	14	3	FP_MUL
IMUL imm32	14	3	FP_MUL
IMUL	15-18	5	
IDIV	56-70	23	
IN/OUT ¹	<225	40	
Jcc ⁶	Not Applicable	0.5	ALU
LOOP	8	1.5	ALU
MOV	0.5	0.5	ALU
MOVSb/MOVSsw	0.5	0.5	ALU
MOVZb/MOVZsw	0.5	0.5	ALU
NEG/NOT/NOP	0.5	0.5	ALU
POP r32	1.5	1	MEM_LOAD,ALU
PUSH	1.5	1	MEM_STORE,ALU
RCL/RCR reg, 1 ⁶	4	1	
RCL/RCR reg, 1 ⁷	4	1	
ROL/ROR	4	1	
RET	8	1	MEM_LOAD,ALU
SAHF	0.5	0.5	ALU
SAL/SAR/SHL/SHR	4	1	
SCAS	4	1.5	ALU, MEM_LOAD
SETcc	5	1.5	ALU
STOSB	5	2	ALU, MEM_STORE
XCHG	1.5	1	ALU

continued

Table C-7 IA-32 General Purpose Instructions (continued)

Instruction	Latency ¹	Throughput	Execution Unit ²
CALL	5	1	ALU, MEM_STORE
MUL	14-18	5	
DIV	56-70	23	

See [“Table Footnotes”](#)

Table Footnotes

The following footnotes refer to all tables in this appendix.

1. Latency information for many of instructions that are complex ($> 4 \mu\text{ops}$) are estimates based on conservative and worst-case estimates. Actual performance of these instructions by the out-of-order core execution unit can range from somewhat faster to significantly faster than the nominal latency data shown in these tables.
2. The names of execution units include: ALU, FP_EXECUTE, FPMOVE, MEM_LOAD, MEM_STORE. See [Figure 1-4](#) for execution units and ports in the out-of-order core. Note the following:
 - The FP_EXECUTE unit is actually a cluster of execution units, roughly consisting of seven separate execution units.
 - The FP_ADD unit handles x87 and SIMD floating-point add and subtract operation.
 - The FP_MUL unit handles x87 and SIMD floating-point multiply operation.
 - The FP_DIV unit handles x87 and SIMD floating-point divide square-root operations.
 - The MMX_SHFT unit handles shift and rotate operations.
 - The MMX_ALU unit handles SIMD integer ALU operations.
 - The MMX_MISC unit handles reciprocal MMX computations and some integer operations.
 - The FP_MISC designates other execution units in port 1 that are separated from the six units listed above.

3. It may be possible to construct repetitive calls to some IA-32 instructions in code sequences to achieve latency that is one or two clock cycles faster than the more realistic number listed in this table.
4. Latency and Throughput of transcendental instructions can vary substantially in a dynamic execution environment. Only an approximate value or a range of values are given for these instructions.
5. The FXCH instruction has 0 latency in code sequences. However, it is limited to an issue rate of one instruction per clock cycle.
6. Selection of conditional jump instructions should be based on the recommendation of section [“Branch Prediction”](#) to improve the predictability of branches. When branches are predicted successfully, the latency of jcc is effectively zero.
7. RCL/RCR with shift count of 1 are optimized. Using RCL/RCR with shift count other than 1 will be executed more slowly.

Latency and Throughput with Memory Operands

Typically, instructions with a memory address as the source operand, add one more μop to the “reg, reg” instructions type listed in Table C-1 through C-7. However, the throughput in most cases remains the same because the load operation utilizes port 2 without affecting port 0 or port 1.

Many IA-32 instructions accept a memory address as either the source operand or as the destination operand. The former is commonly referred to as a load operation, while the latter a store operation.

The latency for IA-32 instructions that perform either a load or a store operation are typically longer than the latency of corresponding register-to-register type of the IA-32 instructions. This is because load or store operations require access to the cache hierarchy and, in some cases, the memory sub-system.

For the sake of simplicity, all data being requested is assumed to reside in the first level data cache (cache hit). In general, IA-32 instructions with load operations that execute in the integer ALU units require two more clock cycles than the corresponding

register-to-register flavor of the same instruction. Throughput of these instructions with load operation remains the same with the register-to-register flavor of the instructions.

Floating-point, MMX technology, Streaming SIMD Extensions and Streaming SIMD Extension 2 instructions with load operations require 6 more clocks in latency than the register-only version of the instructions, but throughput remains the same.

When store operations are on the critical path, their results can generally be forwarded to a dependent load in as few as zero cycles. Thus, the latency to complete and store isn't relevant here.

Stack Alignment



This appendix details on the alignment of the stacks of data for Streaming SIMD Extensions and Streaming SIMD Extensions 2.

Stack Frames

This section describes the stack alignment conventions for both `esp`-based (normal), and `ebp`-based (debug) stack frames. A stack frame is a contiguous block of memory allocated to a function for its local memory needs. It contains space for the function's parameters, return address, local variables, register spills, parameters needing to be passed to other functions that a stack frame may call, and possibly others. It is typically delineated in memory by a stack frame pointer (`esp`) that points to the base of the frame for the function and from which all data are referenced via appropriate offsets. The convention on IA-32 is to use the `esp` register as the stack frame pointer for normal optimized code, and to use `ebp` in place of `esp` when debug information must be kept. Debuggers use the `ebp` register to find the information about the function via the stack frame.

It is important to ensure that the stack frame is aligned to a 16-byte boundary upon function entry to keep local `__m128` data, parameters, and `xmm` register spill locations aligned throughout a function invocation. The Intel C++ Compiler for Win32* Systems supports conventions presented here help to prevent memory references from incurring penalties due to misaligned data by keeping them aligned to 16-byte boundaries. In addition, this scheme supports improved alignment for `__m64` and `double` type data by enforcing that these 64-bit data items are at least eight-byte aligned (they will now be 16-byte aligned).

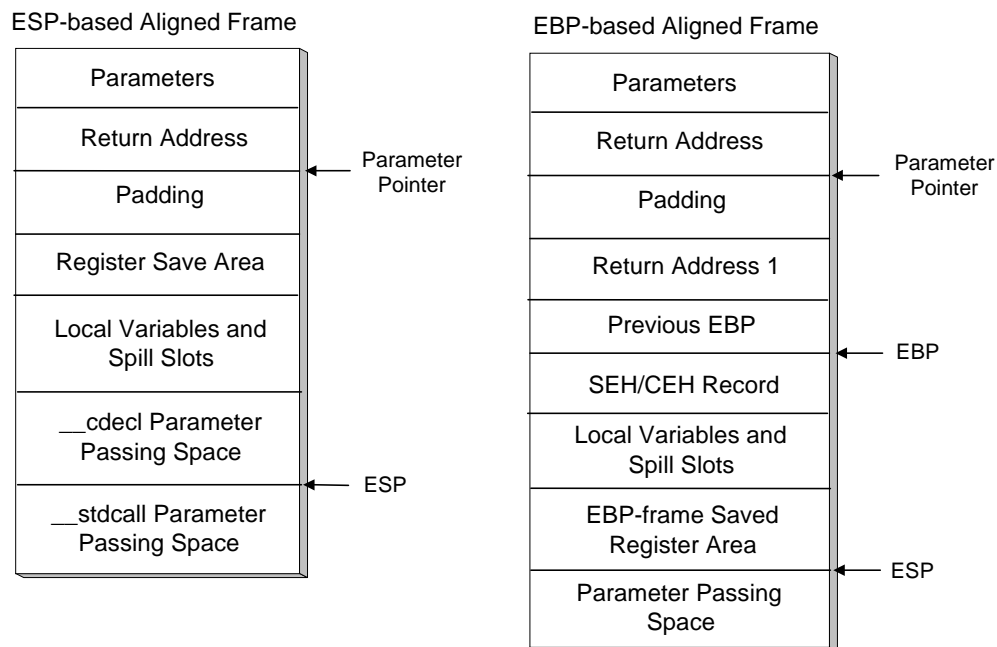
For variables allocated in the stack frame, the compiler cannot guarantee the base of the variable is aligned unless it also ensures that the stack frame itself is 16-byte aligned. Previous IA-32 software conventions, as implemented in most compilers, only

ensure that individual stack frames are 4-byte aligned. Therefore, a function called from a Microsoft*-compiled function, for example, can only assume that the frame pointer it used is 4-byte aligned.

Earlier versions of the Intel C++ Compiler for Win32 Systems have attempted to provide 8-byte aligned stack frames by dynamically adjusting the stack frame pointer in the prologue of `main` and preserving 8-byte alignment of the functions it compiles. This technique is limited in its applicability for the following reasons:

- The `main` function must be compiled by the Intel C++ Compiler.
- There may be no functions in the call tree compiled by some other compiler (as might be the case for routines registered as callbacks).
- Support is not provided for proper alignment of parameters.

The solution to this problem is to have the function's entry point assume only 4-byte alignment. If the function has a need for 8-byte or 16-byte alignment, then code can be inserted to dynamically align the stack appropriately, resulting in one of the stack frames shown in [Figure D-1](#).

Figure D-1 Stack Frames Based on Alignment Type

As an optimization, an alternate entry point can be created that can be called when proper stack alignment is provided by the caller. Using call graph profiling of the VTune analyzer, calls to the normal (unaligned) entry point can be optimized into calls to the (alternate) aligned entry point when the stack can be proven to be properly aligned. Furthermore, a function alignment requirement attribute can be modified throughout the call graph so as to cause the least number of calls to unaligned entry points. As an example of this, suppose function F has only a stack alignment requirement of 4, but it calls function G at many call sites, and in a loop. If G's alignment requirement is 16, then by promoting F's alignment requirement to 16, and making all calls to G go to its aligned entry point, the compiler can minimize the number of times that control passes through the unaligned entry points. [Example D-1](#) and [Example D-2](#) in the following sections illustrate this technique. Note the entry points `foo` and `foo.aligned`, the latter is the alternate aligned entry point.

Aligned `esp`-Based Stack Frames

This section discusses data and parameter alignment and the `declspec(aligned)` extended attribute, which can be used to request alignment in C and C++ code. In creating `esp`-based stack frames, the compiler adds padding between the return address and the register save area as shown in [Example 3-9](#). This frame can be used only when debug information is not requested, there is no need for exception handling support, inlined assembly is not used, and there are no calls to `alloca` within the function.

If the above conditions are not met, an aligned `ebp`-based frame must be used. When using this type of frame, the sum of the sizes of the return address, saved registers, local variables, register spill slots, and parameter space must be a multiple of 16 bytes. This causes the base of the parameter space to be 16-byte aligned. In addition, any space reserved for passing parameters for `stdcall` functions also must be a multiple of 16 bytes. This means that the caller needs to clean up some of the stack space when the size of the parameters pushed for a call to a `stdcall` function is not a multiple of 16. If the caller does not do this, the stack pointer is not restored to its pre-call value.

In [Example D-1](#), we have 12 bytes on the stack after the point of alignment from the caller: the return pointer, `ebx` and `edx`. Thus, we need to add four more to the stack pointer to achieve alignment. Assuming 16 bytes of stack space are needed for local variables, the compiler adds $16 + 4 = 20$ bytes to `esp`, making `esp` aligned to a 0 mod 16 address.

Example D-1 Aligned esp-Based Stack Frames

```
void _cdecl foo (int k)
{
    int j;
    foo:                                     // See Note A
        push    ebx
        mov     ebx, esp
        sub     esp, 0x00000008
        and     esp, 0xffffffff0
        add     esp, 0x00000008
        jmp     common
foo.aligned:
        push    ebx
        mov     ebx, esp

common:                                     // See Note B
        push    edx
        sub     esp, 20
        j = k;
        mov     edx, [ebx + 8]
        mov     [esp + 16], edx
foo(5);
        mov     [esp], 5
        call    foo.aligned
return j;
        mov     eax, [esp + 16]
        add     esp, 20
        pop     edx
        mov     esp, ebx
        pop     ebx
        ret
```



NOTE. *A. Aligned entry points assume that parameter block beginnings are aligned. This places the stack pointer at a 12 mod 16 boundary, as the return pointer has been pushed. Thus, the unaligned entry point must force the stack pointer to this boundary.*

B. The code at the common label assumes the stack is at an 8 mod 16 boundary, and adds sufficient space to the stack so that the stack pointer is aligned to a 0 mod 16 boundary.

Aligned `ebp`-Based Stack Frames

In `ebp`-based frames, padding is also inserted immediately before the return address. However, this frame is slightly unusual in that the return address may actually reside in two different places in the stack. This occurs whenever padding must be added and exception handling is in effect for the function. [Example D-2](#) shows the code generated for this type of frame. The stack location of the return address is aligned 12 mod 16. This means that the value of `ebp` always satisfies the condition $(\text{ebp} \ \& \ 0x0f) == 0x08$. In this case, the sum of the sizes of the return address, the previous `ebp`, the exception handling record, the local variables, and the spill area must be a multiple of 16 bytes. In addition, the parameter passing space must be a multiple of 16 bytes. For a call to a `stdcall` function, it is necessary for the caller to reserve some stack space if the size of the parameter block being pushed is not a multiple of 16.

Example D-2 Aligned ebp-based Stack Frames

```

void _stdcall foo (int k)
{
    int j;
foo:
    push    ebx
    mov     ebx, esp
    sub     esp, 0x00000008
    and     esp, 0xffffffff0
    add     esp, 0x00000008          // esp is (8 mod 16) after add
    jmp     common

foo.aligned:
    push    ebx                    // esp is (8 mod 16) after push
    mov     ebx, esp
common:
    push    ebp                    // this slot will be used for
                                // duplicate return pt
    push    ebp                    // esp is (0 mod 16) after push
                                // (rtn,ebx,ebp,ebp)
    mov     ebp, [ebx + 4]          // fetch return pointer and store
    mov     [esp + 4], ebp          // relative to ebp
                                // (rtn,ebx,rtn,ebp)
    mov     ebp, esp                // ebp is (0 mod 16)
    sub     esp, 28                 // esp is (4 mod 16)
                                //see Note A
    push    edx                    // esp is (0 mod 16) after push
                                // the goal is to make esp and ebp
                                // (0 mod 16) here

```

continued

Example D-2 Aligned ebp-based Stack Frames (continued)

```
j = k;
    mov     edx, [ebx + 8]           // k is (0 mod 16) if caller aligned
                                     // its stack
    mov     [ebp - 16], edx         // J is (0 mod 16)
foo(5);
    add     esp, -4                 // normal call sequence to
                                     // unaligned entry
    mov     [esp], 5
    call    foo                    // for stdcall, callee
                                     // cleans up stack
foo.aligned(5);
    add     esp, -16                // aligned entry, this should
                                     // be a multiple of 16
    mov     [esp], 5
    call    foo.aligned
    add     esp, 12                 // see Note B
return j;
    mov     eax, [ebp-16]
    pop     edx
    mov     esp, ebp
    pop     ebp
    mov     esp, ebx
    pop     ebx
ret 4
}
```



NOTE. *A. Here we allow for local variables. However, this value should be adjusted so that, after pushing the saved registers, esp is $0 \bmod 16$.*

B. Just prior to the call, esp is $0 \bmod 16$. To maintain alignment, esp should be adjusted by 16. When a callee uses the stdcall calling sequence, the stack pointer is restored by the callee. The final addition of 12 compensates for the fact that only 4 bytes were passed, rather than 16, and thus the caller must account for the remaining adjustment.

Stack Frame Optimizations

The Intel C++ Compiler provides certain optimizations that may improve the way aligned frames are set up and used. These optimizations are as follows:

- If a procedure is defined to leave the stack frame 16-byte-aligned and it calls another procedure that requires 16-byte alignment, then the callee's aligned entry point is called, bypassing all of the unnecessary aligning code.
- If a static function requires 16-byte alignment, and it can be proven to be called only by other functions that require 16-byte alignment, then that function will not have any alignment code in it. That is, the compiler will not use `ebx` to point to the argument block and it will not have alternate entry points, because this function will never be entered with an unaligned frame.

Inlined Assembly and `ebx`

When using aligned frames, the `ebx` register generally should not be modified in inlined assembly blocks since `ebx` is used to keep track of the argument block. Programmers may modify `ebx` only if they do not need to access the arguments and provided they save `ebx` and restore it before the end of the function (since `esp` is restored relative to `ebx` in the function's epilog).

For additional information on the use of `ebx` in inline assembly code and other related issues, see relevant application notes in the Intel Architecture Performance Training Center.



CAUTION. *Do not use the `ebx` register in inline assembly functions that use dynamic stack alignment for `double`, `__m64`, and `__m128` local variables unless you save and restore `ebx` each time you use it. The Intel C++ Compiler uses the `ebx` register to control alignment of variables of these types, so the use of `ebx`, without preserving it, will cause unexpected program execution.*

Mathematics of Prefetch Scheduling Distance



This appendix discusses how far away to insert prefetch instructions. It presents a mathematical model allowing you to deduce a simplified equation which you can use for determining the prefetch scheduling distance (PSD) for your application.

For your convenience, the first section presents this simplified equation; the second section provides the background for this equation: the mathematical model of the calculation.

Simplified Equation

A simplified equation to compute PSD is as follows:

$$psd = \left\lceil \frac{Nlookup + Nxfer \cdot (N_{pref} + N_{st})}{CPI \cdot N_{inst}} \right\rceil$$

where

psd is prefetch scheduling distance.

$Nlookup$ is the number of clocks for lookup latency. This parameter is system-dependent. The type of memory used and the chipset implementation affect its value.

$Nxfer$ is the number of clocks to transfer a cache-line. This parameter is implementation-dependent.

N_{pref} and N_{st} are the numbers of cache lines to be prefetched and stored.

CPI is the number of clocks per instruction. This parameter is implementation-dependent.

N_{inst} is the number of instructions in the scope of one loop iteration.

Consider the following example of a heuristic equation assuming that parameters have the values as indicated:

$$psd = \left\lceil \frac{60 + 25 \cdot (N_{pref} + N_{st})}{1.5 \cdot N_{inst}} \right\rceil$$

where 60 corresponds to N_{lookup} , 25 to N_{xfer} , and 1.5 to CPI .

The values of the parameters in the equation can be derived from the documentation for memory components and chipsets as well as from vendor datasheets.



CAUTION. *The values in this example are for illustration only and do not represent the actual values for these parameters. The example is provided as a “starting point approximation” of calculating the prefetch scheduling distance using the above formula. Experimenting with the instruction around the “starting point approximation” may be required to achieve the best possible performance.*

Mathematical Model for PSD

The parameters used in the mathematics discussed are as follows:

psd	prefetch scheduling distance (measured in number of iterations)
il	iteration latency
T_c	computation latency per iteration with prefetch caches
T_l	memory leadoff latency including cache miss latency, chip set latency, bus arbitration, etc.
T_b	data transfer latency which is equal to number of lines per iteration * line burst latency

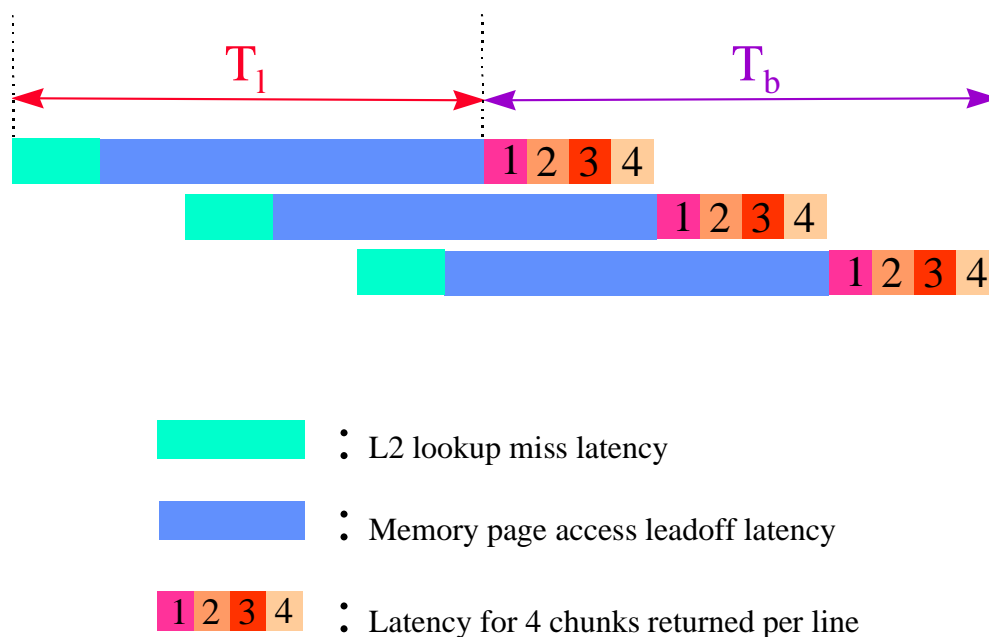
Note that the potential effects of μop reordering are not factored into the estimations discussed.

Examine [Example E-1](#) that uses the `prefetchnta` instruction with a prefetch scheduling distance of 3, that is, $psd = 3$. The data prefetched in iteration i , will actually be used in iteration $i+3$. T_c represents the cycles needed to execute `top_loop` - assuming all the memory accesses hit L1 while il (iteration latency) represents the cycles needed to execute this loop with actually run-time memory footprint. T_c can be determined by computing the critical path latency of the code dependency graph. This work is quite arduous without help from special performance characterization tools or compilers. A simple heuristic for estimating the T_c value is to count the number of instructions in the critical path and multiply the number with an artificial CPI. A reasonable CPI value would be somewhere between 1.0 and 1.5 depending on the quality of code scheduling.

Example E-1 Calculating Insertion for Scheduling Distance of 3

```
top_loop:
    prefetchnta [edx+esi+32*3]
    prefetchnta [edx*4+esi+32*3]
    . . . . .
    movaps xmm1, [edx+esi]
    movaps xmm2, [edx*4+esi]
    movaps xmm3, [edx+esi+16]
    movaps xmm4, [edx*4+esi+16]
    . . . . .
    . . .
    add esi, 32
    cmp esi, ecx
    jl top_loop
```

Memory access plays a pivotal role in prefetch scheduling. For more understanding of a memory subsystem, consider Streaming SIMD Extensions and Streaming SIMD Extensions 2 memory pipeline depicted in [Figure E-1](#).

Figure E-1 Pentium II, Pentium III and Pentium 4 Processors Memory Pipeline Sketch

Assume that three cache lines are accessed per iteration and four chunks of data are returned per iteration for each cache line. Also assume these 3 accesses are pipelined in memory subsystem. Based on these assumptions,
 $T_b = 3 * 4 = 12$ FSB cycles.

T_1 varies dynamically and is also system hardware-dependent. The static variants include the core-to-front-side-bus ratio, memory manufacturer and memory controller (chipset). The dynamic variants include the memory page open/miss occasions, memory accesses sequence, different memory types, and so on.

To determine the proper prefetch scheduling distance, follow these steps and formulae:

- Optimize T_c as much as possible

- Use the following set of formulae to calculate the proper prefetch scheduling distance:

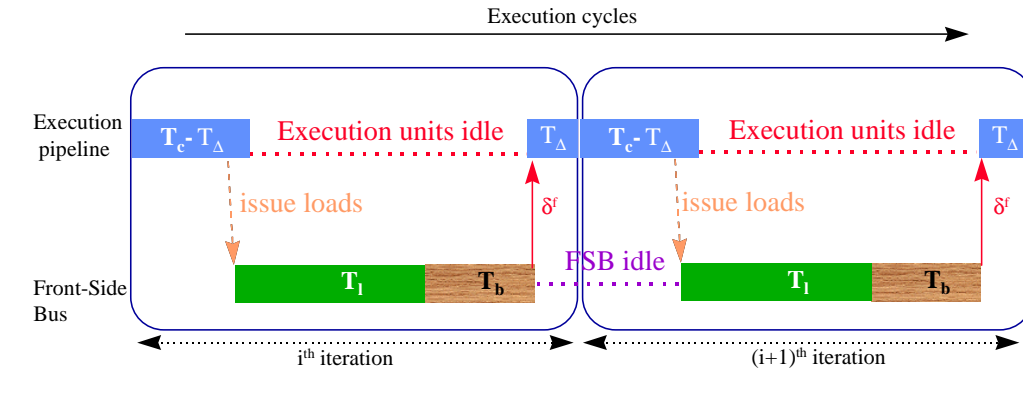
$$\begin{array}{lll}
 T_c \geq T_l + T_b & psd = 1 & il = T_c \\
 T_l + T_b > T_c > T_b & psd = \left\lceil \frac{T_l + T_b}{T_c} \right\rceil & il = T_c \\
 T_b \geq T_c & psd = 1 + \left\lceil \frac{T_l}{T_b} \right\rceil & il = T_b
 \end{array}$$

- Schedule the prefetch instructions according to the computed prefetch scheduling distance.
- For optimized memory performance, apply techniques described in “[Memory Optimization Using Prefetch](#)” in Chapter 6.

The following sections explain and illustrate the architectural considerations involved in the prefetch scheduling distance formulae above.

No Preloading or Prefetch

The traditional programming approach does not perform data preloading or prefetch. It is sequential in nature and will experience stalls because the memory is unable to provide the data immediately when the execution pipeline requires it. Examine [Figure E-2](#).

Figure E-2 Execution Pipeline, No Preloading or Prefetch

As you can see from [Figure E-2](#), the execution pipeline is stalled while waiting for data to be returned from memory. On the other hand, the front side bus is idle during the computation portion of the loop. The memory access latencies could be hidden behind execution if data could be fetched earlier during the bus idle time.

Further analyzing Figure 6-10,

- assume execution cannot continue till last chunk returned and
- δ^f indicates flow data dependency that stalls the execution pipelines

With these two things in mind the iteration latency (il) is computed as follows:

$$il \cong T_c + T_l + T_b$$

The iteration latency is approximately equal to the computation latency plus the memory leadoff latency (includes cache miss latency, chipset latency, bus arbitration, and so on.) plus the data transfer latency where

transfer latency = number of lines per iteration * line burst latency.

This means that the decoupled memory and execution are ineffective to explore the parallelism because of flow dependency. That is the case where prefetch can be useful by removing the bubbles in either the execution pipeline or the memory pipeline.

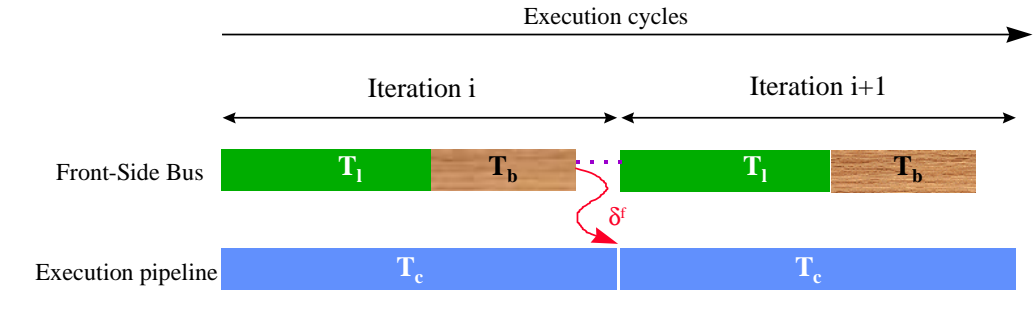
With an ideal placement of the data prefetching, the iteration latency should be either bound by execution latency or memory latency, that is

$$il = \text{maximum}(T_c, T_b).$$

Compute Bound (Case: $T_c \geq T_l + T_b$)

Figure E-3 represents the case when the compute latency is greater than or equal to the memory leadoff latency plus the data transfer latency. In this case, the prefetch scheduling distance is exactly 1, i.e. prefetch data one iteration ahead is good enough. The data for loop iteration i can be prefetched during loop iteration $i-1$, the δ^f symbol between front-side bus and execution pipeline indicates the data flow dependency.

Figure E-3 Compute Bound Execution Pipeline



The following formula shows the relationship among the parameters:

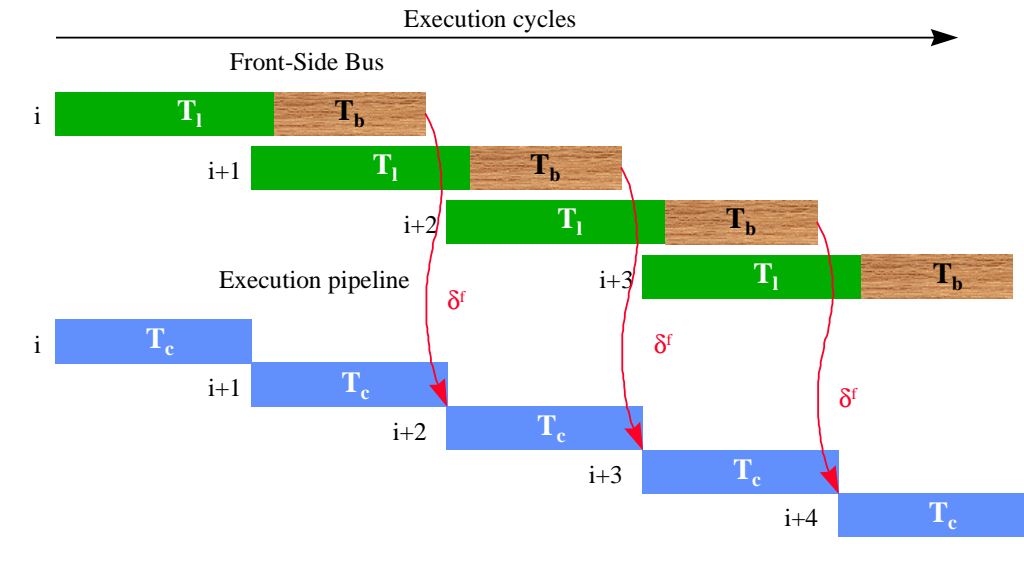
$$psd = \left\lceil \frac{T_l + T_b}{T_c} \right\rceil \equiv 1 \quad il = T_c$$

It can be seen from this relationship that the iteration latency is equal to the computation latency, which means the memory accesses are executed in background and their latencies are completely hidden.

Compute Bound (Case: $T_l + T_b > T_c > T_b$)

Now consider the next case by first examining [Figure E-4](#).

Figure E-4 Compute Bound Execution Pipeline



For this particular example the prefetch scheduling distance is greater than 1. Data being prefetched for iteration i will be consumed in iteration $i+2$.

Figure 6-12 represents the case when the leadoff latency plus data transfer latency is greater than the compute latency, which is greater than the data transfer latency. The following relationship can be used to compute the prefetch scheduling distance.

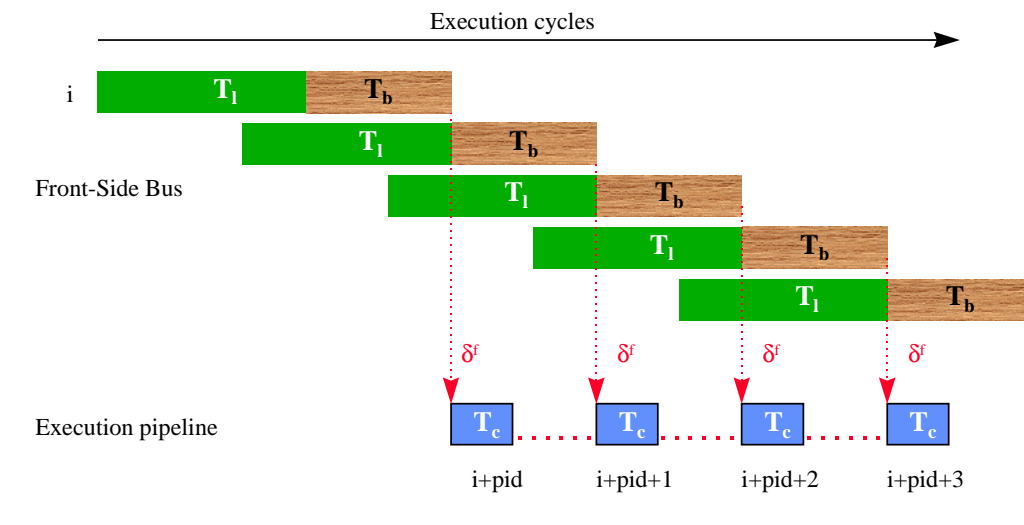
$$psd = \left\lceil \frac{T_l + T_b}{T_c} \right\rceil > 1 \quad il = T_c$$

In consequence, the iteration latency is also equal to the computation latency, that is, compute bound program.

Memory Throughput Bound (Case: $T_b \geq T_c$)

When the application or loop is memory throughput bound, the memory latency is no way to be hidden. Under such circumstances, the burst latency is always greater than the compute latency. Examine [Figure E-5](#).

Figure E-5 Memory Throughput Bound Pipeline



The following relationship calculates the prefetch scheduling distance (or prefetch iteration distance) for the case when memory throughput latency is greater than the compute latency.

$$psd = \left\lceil \frac{T_l + T_b}{T_b} \right\rceil = 1 + \left\lceil \frac{T_l}{T_b} \right\rceil > 1 \quad \text{if } T_b \geq T_c$$

Apparently, the iteration latency is dominant by the memory throughput and you cannot do much about it. Typically, data copy from one space to another space, for example, graphics driver moving data from writeback memory to you cannot do much

about it. Typically, data copy from one space to another space, for example, graphics driver moving data from writeback memory to write-combining memory, belongs to this category, where performance advantage from prefetch instructions will be marginal.

Example

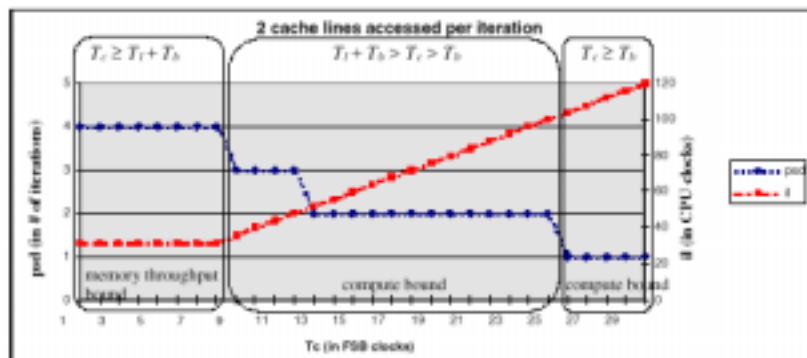
As an example of the previous cases consider the following conditions for computation latency and the memory throughput latencies. Assume $T_l = 18$ and $T_b = 8$ (in front side bus cycles).

$$\text{if } T_c \geq 26 \Rightarrow psd = \left\lceil \frac{18+8}{T_c} \right\rceil = 1$$

$$\text{if } 26 > T_c > 8 \Rightarrow 2 \leq psd = \left\lceil \frac{18+8}{T_c} \right\rceil \leq 3$$

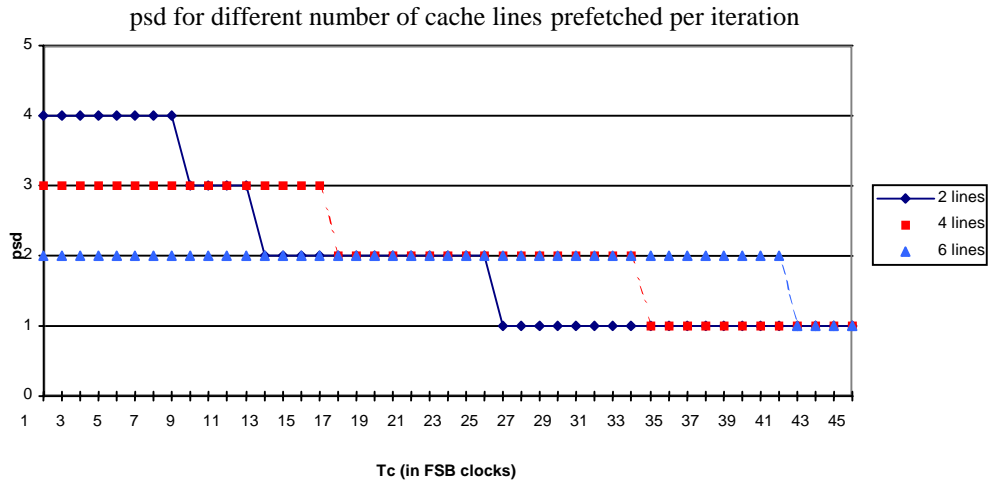
$$\text{if } T_c \leq 8 \Rightarrow psd = 1 + \left\lceil \frac{18}{8} \right\rceil = 4$$

Now for the case $T_l = 18$, $T_b = 8$ (2 cache lines are needed per iteration) examine the following graph. Consider the graph of accesses per iteration in example 1, [Figure E-6](#).

Figure E-6 Accesses per Iteration, Example 1

The prefetch scheduling distance is a step function of T_c , the computation latency. The steady state iteration latency (il) is either memory-bound or compute-bound depending on T_c if prefetches are scheduled effectively.

The graph in example 2 of accesses per iteration in [Figure E-7](#) shows the results for prefetching multiple cache lines per iteration. The cases shown are for 2, 4, and 6 cache lines per iteration, resulting in differing burst latencies. ($T_i=18$, $T_b=8, 16, 24$).

Figure E-7 Accesses per Iteration, Example 2

In reality, the front-side bus (FSB) pipelining depth is limited, that is, only four transactions are allowed at a time in the Pentium III and Pentium 4 processors. Hence a transaction bubble or gap, T_g , (gap due to idle bus of imperfect front side bus pipelining) will be observed on FSB activities. This leads to consideration of the transaction gap in computing the prefetch scheduling distance. The transaction gap, T_g , must be factored into the burst cycles, T_b , for the calculation of prefetch scheduling distance.

The following relationship shows computation of the transaction gap.

$$T_g = \max(T_l - c * (n - 1), 0)$$

where T_l is the memory leadoff latency, c is the number of chunks per cache line and n is the FSB pipelining depth.

Index

A

- absolute difference of signed numbers, 4-22
- absolute difference of unsigned numbers, 4-22
- absolute value, 4-24
- accesses per iteration, E-11, E-12
- algorithm to avoid changing the rounding mode, 2-28, 2-31, 2-47
- aligned ebp-based frame, D-4, D-6
- aligned esp-based stack frames, D-4
- Alignment, 2-22
 - stack, 2-34
 - coe, 2-40
- AoS format, 3-26
- application performance tools, A-1
- Arrays, Aligning, 2-31
- assembly coach, A-18
- assembly coach techniques, A-18
- automatic processor dispatch support, A-3
- automatic vectorization, 3-15, 3-16

B

- Branch Prediction, 2-12
- branch prediction, 2-4

C

- cache blocking techniques, 6-28
- cache hierarchy, A-11
- cache level, 6-4

- cache management
 - simple memory copy, 6-38
 - video decoder, 6-38
 - video encoder, 6-37
- cache performance, A-11
- calculating insertion for scheduling distance, E-3
- call graph profiling, A-12
- Call Graph view, A-12
- call information, A-14
- changing the rounding mode, 2-47
- checking for MMX technology support, 3-2
- checking for Streaming SIMD Extensions support, 3-3
- child function, A-14
- classes (C/C++), 3-14
- clipping to an arbitrary signed range, 4-24
- clipping to an arbitrary unsigned range, 4-27
- code coach, A-15, A-17
- code optimization advice, A-15, A-18
- code optimization options, A-3
- Code segment, Data in, 2-37
- coding methodologies, 3-10
- coding techniques, 3-10
 - absolute difference of signed numbers, 4-22
 - absolute difference of unsigned numbers, 4-22
 - absolute value, 4-24
 - clipping to an arbitrary signed range, 4-24
 - clipping to an arbitrary unsigned range, 4-27
 - generating constants, 4-20
 - interleaved pack with saturation, 4-7
 - interleaved pack without saturation, 4-9
 - non-interleaved unpack, 4-10

- coding techniques (cont.)
 - signed unpack, 4-6
 - simplified clipping to an arbitrary signed range, 4-26
 - unsigned unpack, 4-5
- coherent requests, 6-11
- command-line options, A-2
 - automatic processor dispatch support, A-3
 - floating-point arithmetic precision, A-5
 - inline expansion of library functions, A-5
 - loop unrolling, A-4
 - rounding control, A-5
 - targeting a processor, A-3
 - vectorizer switch, A-4
- comparing register values, 2-58
- compiler intrinsics
 - _mm_load, 6-2, 6-37
 - _mm_prefetch, 6-2, 6-37
- compiler intrinsics, _mm_stream, 6-2, 6-37
- compiler plug-in, A-2
- compiler-supported alignment, 3-21
- complex instructions, 2-53
- computation latency, E-7
- computation-intensive code, 3-9
- compute bound, E-7, E-8
- converting code to MMX technology, 3-6
- counters, A-11
- CPUID instruction, 3-2

D

- Data, Code segment and, 2-37
- data alignment, 3-16
- data arrangement, 5-4
- data copy, E-10
- data deswizzling, 5-11, 5-12
- Data structures
 - Access pattern versus alignment, 2-33
 - Aligning, 2-31
- data swizzling, 5-7
- data swizzling using intrinsics, 5-9

- debug symbols, A-14
- decoupled memory, E-6
- divide instructions, 2-55

E

- EBS. See event-based sampling
- eliminating branches, 2-12, 2-14
- EMMS instruction, 4-2, 4-3, 4-4
- event-based sampling, A-9
- extract word instruction, 4-12

F

- fist instruction, 2-46
- fldcw instruction, 2-46
- floating-point applications, 2-41
- floating-point arithmetic precision options, A-5
- floating-point code
 - improving parallelism, 2-49
 - loop unrolling, 2-20
 - memory access stall information, 2-29
 - memory operands, 2-51
 - operations with integer operands, 2-52
 - optimizing, 2-41
 - transcendental functions, 2-52
- floating-point operations with integer operands, 2-52
- floating-point stalls, 2-51
- flow dependency, E-7
- flush to zero, 5-19
- FXCH instruction, 2-51

G

- general optimization techniques, 2-1
 - branch prediction, 2-12
 - static prediction, 2-15
- generating constants, 4-20

H

horizontal computations, 5-15
hotspots, 3-8, A-15, A-16

I

increasing bandwidth of memory fills, 4-34
increasing bandwidth of video fills, 4-34
indirect branch, 2-20
inline assembly, 4-5
inline expansion of library functions option, A-5
inlined assembly blocks, D-9
inlined-asm, 3-12
insert word instruction, 4-13
instruction scheduling, 2-36, 4-36
instruction selection, 2-52
integer and floating-point multiply, 2-55
integer divide, 2-55
integer-intensive application, 4-1
Intel® Performance Library Suite, A-1
interleaved pack with saturation, 4-7
interleaved pack without saturation, 4-9
interprocedural optimization, A-6
IPO. See interprocedural optimization

L

large load stalls, 2-29
latency, 2-51, 6-3
lea instruction, 2-53
loading and storing to and from the same DRAM page, 4-35
loop blocking, 3-30
loop unrolling, 2-20
loop unrolling option, A-4, A-5

M

memory bank conflicts, 6-3
memory O=optimization U=using P=prefetch, 6-16
memory operands, 2-51
memory optimization, 4-31
memory optimizations
 loading and storing to and from the same DRAM page, 4-35
 partial memory accesses, 4-32
 using aligned stores, 4-35
memory performance, 3-23
memory reference instructions, 2-57
memory throughput bound, E-9
minimizing prefetches number, 6-23
misaligned data access, 3-17
misalignment in the FIR filter, 3-18
move byte mask to integer, 4-15
MOVQ Instruction, 4-35

N

new SIMD-integer instructions
 extract word, 4-12
 insert word, 4-13
 move byte mask to integer, 4-15
 packed average byte or word), 4-29
 packed multiply high unsigned, 4-28
 packed shuffle word, 4-17
 packed signed integer word maximum, 4-28
 packed sum of absolute differences, 4-28
Newton-Raphson iteration, 5-2
non-coherent requests, 6-11
non-interleaved unpack, 4-10
non-temporal stores, 6-36
NOPs, 2-64
 To align instructions, 2-64
 XCHG EAX,EAX
 Special hardware support for, 2-65
numeric exceptions
 flush to zero, 5-19

O

- optimizing cache utilization
 - cache management, 6-37
 - examples, 6-13
 - non-temporal store instructions, 6-8
 - prefetch and load, 6-7
 - prefetch Instructions, 6-6
 - prefetching, 6-5
 - SFENCE instruction, 6-13, 6-14
 - streaming, non-temporal stores, 6-8
- optimizing floating-point applications
 - copying, shuffling, 5-15
 - data arrangement, 5-4
 - data deswizzling, 5-11
 - data swizzling using intrinsics, 5-9
 - horizontal ADD, 5-16
 - planning considerations, 5-2
 - rules and suggestions, 5-1
 - scalar code, 5-3
 - vertical versus horizontal computation, 5-4
- optimizing floating-point code, 2-41

P

- pack instruction, 4-9
- pack instructions, 4-7
- packed average byte or word, 4-29
- packed multiply high unsigned, 4-28
- packed shuffle word, 4-17
- packed signed integer word maximum, 4-28
- packed sum of absolute differences, 4-28
- pairing, A-14
- parallelism, 3-10, E-6
- parameter alignment, D-4
- parent function, A-14
- partial memory accesses, 4-32
- PAVGB instruction, 4-29
- PAVGW instruction, 4-29
- penalties, A-14
- performance counter events, A-9
- Performance Library Suite, A-19
 - architecture, A-20
 - Image Processing Library, A-19
 - Math Kernel Library, A-19
 - optimizations, A-21
 - Recognition Primitives Library, A-19
 - Signal Processing Library, A-19
- PEXTRW instruction, 4-12
- PGO. See profile-guided optimization
- PINSRW instruction, 4-13
- PLS. See Performance Library Suite
- PMINSW instruction, 4-28
- PMINUB instruction, 4-28
- PMOVMASKB instruction, 4-15
- PMULHUW instruction, 4-28
- predictable memory access patterns, 6-5
- prefetch and cacheability Instructions, 6-3
- prefetch and load Instructions, 6-7
- prefetch concatenation, 6-21, 6-22
- prefetch instruction, 6-1
- prefetch instruction considerations, 6-19
 - cache blocking techniques, 6-28
 - concatenation, 6-21
 - minimizing prefetches number, 6-23
 - no preloading or prefetch, E-5
 - prefetch scheduling distance, E-4
 - scheduling distance, 6-19
 - single-pass execution, 6-3, 6-33
 - spread prefetch with computation instructions, 6-26
 - strip-mining, 6-31
- prefetch instructions, 6-5
- prefetch scheduling distance, 6-19, E-5, E-7, E-9
- prefetch use
 - predictable memory access patterns, 6-5
 - time-consuming innermost loops, 6-5
- prefetching, A-4
- prefetching concept, 6-4
- prefetchnta instruction, 6-30
- profile-guided optimization, A-6
- prolog sequences, 2-60

PSADBW instruction, 4-28

PSHUF instruction, 4-17

R - S

reciprocal instructions, 5-2

rounding control option, A-5

sampling, A-7

- event-based, A-9

- time-based, A-8

Self-modifying code, 2-36

SFENCE Instruction, 6-13, 6-14

signed unpack, 4-6

SIMD integer code, 4-2

SIMD-floating-point code, 5-1

simple memory copy, 6-38

simplified 3D geometry pipeline, 6-17

simplified clipping to an arbitrary signed range, 4-26

single-pass versus multi-pass execution, 6-33

SoA format, 3-26

software write-combining, 6-36

spread prefetch, 6-27

Spreadsheet, A-12

Stack Alignment

- Example of dynamic, 2-34

Stack alignment, 2-34

stack alignment, 3-19

stack frame, D-1

stack frame optimization, D-9

static assembly analyzer, A-15

static branch prediction algorithm, 2-16

static code analysis, A-14

static prediction, 2-15

static prediction algorithm, 2-15

streaming store, 6-38

streaming stores

- coherent requests, 6-11

- non-coherent requests, 6-11

strip mining, 3-28, 3-30

strip-mining, 6-31, 6-32

Structs, Aligning, 2-31

swizzling data. See data swizzling.

T

targeting a processor option, A-3

TBS. See time-based sampling

time-based sampling, A-7, A-8

time-consuming innermost loops, 6-5

TLB. See transaction lookaside buffer

transaction lookaside buffer, 6-39

transcendental functions, 2-52

transfer latency, E-7, E-8

tuning application, A-7

U

unpack instructions, 4-10

unsigned unpack, 4-5

using MMX code for copy or shuffling functions, 5-15

V

vector class library, 3-15

vectorization, 3-10

vectorized code, 3-15

vectorizer switch options, A-4

vertical versus horizontal computation, 5-4

View by Call Sites, A-12, A-14

VTune analyzer, 3-8, A-1

VTune™ Performance Analyzer, 3-8

W

write-combining buffer, 6-36

write-combining memory, 6-36